Algorithms for Distributed Data Structures and Self-Stabilizing Overlay Networks

Dissertation

In partial fulfillment of the requirements for the academic degree
Doctor rerum naturalium (Dr. rer. nat.)

Faculty of Computer Science,
Electrical Engineering and Mathematics
Department of Computer Science
Research Group Theory of Distributed Systems

Michael Feldmann

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
Jun. Prof. Dr. Sevag Gharibian	Paderborn University
Dr. Matthias Fischer	Paderborn University
Dr. Peter Pfahler	Paderborn University

Contact:
Michael Feldmann (michael.feldmann@upb.de)

Abstract

This thesis considers the realization of distributed data structures and the construction of distributed protocols for self-stabilizing overlay networks.

In the first part of this thesis, we provide distributed protocols for queues, stacks and priority queues that serve the insertion and deletion of elements within a logarithmic amount of rounds. Our protocols respect semantic constraints such as sequential consistency or serializability and the individual semantic constraints given by the type (queue, stack, priority queue) of the data structure. We furthermore provide a protocol that handles joining and leaving nodes. As an important side product, we present a novel protocol solving the distributed k-selection problem in a logarithmic amount of rounds, that is, to find the k-smallest elements among a polynomial number of elements spread among n nodes.

The second part of this thesis is devoted to the construction of protocols for self-stabilizing overlay networks, i.e., distributed protocols that transform an overlay network from any initial (potentially illegitimate) state into a legitimate state in finite time. We present protocols for self-stabilizing generalized De Bruijn graphs, self-stabilizing quadtrees and self-stabilizing supervised skip rings. Each of those protocols comes with unique properties that makes it interesting for certain distributed applications. Generalized De Bruijn networks provide routing within a constant amount of hops, thus serving the interest in networks that require a low latency for requests. The protocol for the quadtree guarantees monotonic searchability as well as a geometric variant of monotonic searchability, making it interesting for wireless networks or applications needed in the area of computational geometry. The supervised skip ring can be used to construct a self-stabilizing publish-subscribe system.

Zusammenfassung

Diese Dissertation befasst sich mit der Realisierung von verteilten Datenstrukturen und der Konstruktion von selbststabilisierenden Overlaynetzen.

Im ersten Teil dieser Dissertation präsentieren wir verteilte Protokolle für Queues, Stacks und Priority Queues, welche Elemente innerhalb einer logarithmischen Anzahl an Runden einfügen bzw. entfernen können. Zusätzlich respektieren unsere Protokolle neben Semantiken wie sequentielle Konsistenz oder Serialisierbarkeit auch Semantiken welche vom Typ (Queue, Stack, Priority Queue) der Datenstruktur abhängen. Wir stellen ebenfalls ein Protokoll vor, welches das Einfügen und Entfernen neuer Knoten realisiert. Ein wichtiges Nebenprodukt stellt ein neues Protokoll dar, welches die verteilte k-Selektion innerhalb einer logarithmischen Anzahl an Runden löst, d.h. es findet das k-kleinste Element unter einer polynomiellen Anzahl von Elementen, die auf n Knoten verteilt sind.

Der zweite Teil dieser Dissertation befasst sich mit der Konstuktion von Protokollen für selbststabilisierende Overlaynetze, d.h. verteilte Protokolle, welche ein Overlaynetz in endlicher Zeit von einem beliebigen initialen Zustand (der potentiell nicht legitim sein muss) in einen legitimen Zustand transformiert. Wir präsentieren Protokolle für selbststabilisierende generalisierte De Bruijn Graphen, selbststabilisierende Quadtrees und selbststabilisierende überwachte Skip-Ringe. Jedes dieser Protokolle ist aufgrund seiner Eigenschaften interessant für spezifische verteilte Anwendungen. Generalisierte De Bruijn Netzwerke erlauben Routing mit einer konstanten Anzahl an Hops und sind daher interessant für Anwendungen bei welchen eine geringe Latenz erforderlich ist. Das Protokoll für Quadtrees erfüllt die monotone Suchbarkeit sowie dessen geometrische Variante und ist daher interessant für Dratlosnetzwerke oder Anwendungen aus dem Bereich der Computational Geometry. Der überwachte Skipring kann zur Konstruktion eines selbststabilisierenden Publish-Subscribe Systems verwendet werden.

Preface

First and foremost I would like to thank my advisor Christian Scheideler, who gave me the opportunity to work in his research group, which is not something I took for granted. Throughout my journey, he was always available to discuss research problems and give his advice.

I would like to thank Friedhelm Meyer auf der Heide, who served as the second reviewer, as well as Sevag Gharibian, Matthias Fischer and Peter Pfahler for serving on the doctoral panel.

I am very grateful to the CRC 901 "On-The-Fly Computing" for funding my research.

Many thanks also go to all of my co-authors who I enjoyed working with on several research papers. I would like to thank all past and current members of the research group "Theory of Distributed Systems". It has been an honor to have you as colleagues. Special thanks go to Marion Hucke, Uli Ahlers, Thomas Thissen and Ulf-Peter Schröder, who keep the group running.

Last but not least, I would like to thank my parents and my brother for always supporting me in what I am doing. Thank you!

Michael Feldmann
January 2021

Contents

CHAPTER 1

Introduction

Ever since the invention of the internet, distributed computing has become more and more relevant to our society and continues to do so. Important applications are, for example, social media networks, streaming platforms, online games and many more. A distributed system is the union of multiple computers (or processes) that communicate via message-passing in order to solve a problem or to achieve a common goal. As the size of distributed systems increases, allowing even multiple millions of processes to participate, algorithms need to scale efficiently w.r.t. the number of participants in order to remain practically relevant. Another important feature of a distributed system is its robustness, i.e., its ability to cope with (external) faults like message loss or adversarial attacks.

The theory of distributed computing provides the groundwork for many of the above mentioned applications. Research areas range from classical distributed problems such as routing, coloring, maximum independent set and maximal matching to more recent ones like blockchains and hybrid networks. From a theoretical perspective, a distributed system is usually abstracted via an *overlay network*, i.e., a graph indicating the connections between the participants of the system. Using this abstraction yields a precise way to describe distributed protocols. One key property of overlay networks is that they are dynamic by nature, meaning that not only the participants of the system can change over the course of time, but also distributed protocols on overlay networks are capable of modifying the connections between the participants.

In this thesis, we present solutions to problems of two different (but close) research areas within the theory of distributed computing, both of them relying on the capabilities of overlay networks.

In the first part we investigate *distributed data structures*. We consider fundamental data structures (queues, stacks and priority queues) and present algorithms for implementing these data structures in a distributed setting. The elements of the data structures are distributed equally among the nodes of the overlay network such that no single node has direct access to all elements. There are a multitude of problems that one needs to solve in order to come up with not only efficient but also semantically correct solutions. Areas of applications for distributed data structures include, among others, scheduling, distributed counting or distributed sorting.

In the second part of this thesis, we then study *self-stabilizing overlay networks*. These are overlay networks that are able to recover their topology only by allowing communication between the nodes and without external intervention. Such systems are particularly useful for increasing the robustness against failures (like message loss or a blackout of nodes) or adversarial attacks. We investigate self-stabilizing protocols for specific topologies (generalized De Bruijn graphs, quadtrees and supervised skip rings), each having its own unique application area.

Thesis Overview

This thesis is split into two separate parts: The first part is dedicated to the study of distributed data structures, while the second part deals with self-stabilizing overlay networks. The first part starts with a chapter that introduces the model and some well-known techniques (see Chapter 2).

Distributed Queues and Stacks. In Chapter 3 we propose distributed protocols for queues and stacks that perform well even when having to deal with a large amount of requests; i.e., we can avoid bottlenecks in high-throughput scenarios when using these protocols. More specifically, our protocols can serve requests on the data structure in $\mathcal{O}(\log n)$ rounds w.h.p.[1], independent of the rate at which new requests are injected into the system. Our protocols work in asynchronous settings and not only provide strong semantics like sequential consistency, but also satisfy semantic constraints specifically tailored to the FIFO/LIFO properties of queues and stacks. To do so, we use an *aggregation tree* to aggregate batches of requests, ensuring that our protocols scale even for a high rate of incoming requests. For the stack we are also able to show that by using a technique called *local combining*, we can reduce the worst-case bound on the message size. Finally, we also introduce protocols for dealing with node dynamics, i.e., handling joining and leaving nodes. Here we show that a joining node is allowed to start generating requests on the data structure after $\mathcal{O}(\log n)$ rounds w.h.p. A node that wants to leave the system can do so after only $\mathcal{O}(1)$ rounds in case no conflicts with other leaving nodes have to be resolved.

Chapter 3 is based on the following publication:

> M. Feldmann, C. Scheideler and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, cf. [FSS18a].

The protocol for the distributed stack (Section 3.4) is based on the full version of the paper above:

> M. Feldmann, C. Scheideler and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue", *CoRR, abs/1802.07504*, 2018, cf. [FSS18b].

Distributed Priority Queues and k-Selection. In Chapter 4 we investigate another fundamental data structure in the distributed setting, namely distributed priority queues. Here we present two different protocols: The first realizes a distributed priority queue for a constant amount of priorities, while the second works for an arbitrary amount. Similar to our distributed implementations for queues and stacks, these protocols work in asynchronous settings and satisfy semantic constraints specifically tailored to mimic the behavior of priority queues in the sequential setting. Both protocols use the same aggregation tree as the protocols for queues and stacks

[1] An event holds with high probability (w.h.p.), if it holds with probability at least $1 - 1/n^c$ where the constant c can be made arbitrarily large.

from the previous chapter. For part of the protocol for arbitrarily many priorities we provide a novel distributed protocol for the k-selection problem that runs in $\mathcal{O}(\log n)$ rounds w.h.p., which is of independent interest. Our protocols serve requests on the data structure in $\mathcal{O}(\log n)$ rounds w.h.p. and also work together well with the protocol from the previous chapter that handles joining and leaving nodes.

Chapter 4 is based on the following publication:

> M. Feldmann and C. Scheideler. "Skeap & Seap: Scalable Distributed Priority Queues for Constant and Arbitrary Priorities". In: *Proceedings of the 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019, cf. [FS19].

We conclude the first part of the thesis in Chapter 5.

The first chapter of the second part (Chapter 6) presents our model alongside with related work and the well-known protocols for self-stabilizing sorted lists and rings.

Self-Stabilizing Generalized De Bruijn Graphs. In Chapter 7 we present a novel protocol BuildGDB, which is able to build a generalized De Bruijn graph out of every initially weakly connected graph in a self-stabilizing manner. The resulting structure has a diameter of d for some constant $d \geq 2$, while the outdegree of nodes is only $\Theta(\sqrt[d]{n})$. This tradeoff between degree and diameter is asymptotically optimal. Our protocol works in a bottom-up manner, by first arranging the nodes in a sorted list, then expanding the neighborhood in the list so that we arrive at a sorted $\sqrt[d]{n}$-connected list. Finally, we use the $\sqrt[d]{n}$-connected list to generate De Bruijn connections that are used for the routing protocol.

We also present a routing protocol for this network that is able to route any message from any source node s to any target node t in exactly d hops w.h.p. Finally, we show that once new nodes join the system, the amount of work for old nodes is within $\Theta(\sqrt[d]{n})$, which is asymptotically optimal. More precisely, each old node only has to build or redirect $\Theta(\sqrt[d]{n})$ edges w.h.p. once the number of nodes in the system increases by a factor of 2^d.

Chapter 7 is based on the following publication:

> M. Feldmann and C. Scheideler. "A Self-stabilizing General De Bruijn Graph". In: *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017, cf. [FS17].

Self-Stabilizing Quad Trees. In Chapter 8 we investigate *monotonic searchability* in a geometric setting; i.e., nodes are now spread on a 2-dimensional plane. We present a novel self-stabilizing protocol BuildQT that arranges all nodes in a *quadtree*, which is a data structure that is frequently used in the area of computational geometry. The protocol can easily be generalized to higher dimensional settings, realizing the first self-stabilizing protocol for *octrees*. We also give a routing protocol SearchQT by which messages are routed among the nodes in a quadtree-like fashion. This means that if the Euclidean distance between any two nodes is at least $1/n$, then any message routed between any two nodes in the quadtree arrives at its destination after

$\mathcal{O}(\log n)$ hops. Interestingly, SearchQT is defined such that monotonic searchability is satisfied; i.e., once a search request generated by a node u successfully reaches its target node v, any search request for v initiated by u thereafter reaches v as well. We show that in our geometric setting SearchQT satisfies an even stronger notion of monotonic searchability, which preserves monotonicity not only for existing target nodes but also for any target coordinate in the 2-dimensional plane. We call this property *geographic monotonic searchability*.

Chapter 8 is based on the following publication:

> M. Feldmann, C. Kolb, and C. 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)*, 2018, cf. [FKS18].

Self-Stabilizing Publish-Subscribe Systems. In Chapter 9 we present two major results. First we describe a novel protocol for a supervised skip ring. In a skip ring the nodes form a sorted ring with additional shortcuts, such that the diameter of the network is only $\lceil \log n \rceil$, while the degree of each node is $\mathcal{O}(\log n)$ in the worst case and constant on average. The protocol relies on the help of a gateway (the *supervisor*) that is commonly known among all nodes. We show that the communication work of the supervisor is low in our protocol.

For the second result we extend the self-stabilizing supervised skip ring to a (topic-based) self-stabilizing publish-subscribe system. Here we are primarily concerned with the self-stabilizing delivery of all publications to all subscribers of a specific topic. We store publications at each node in a Patricia trie and provide a self-stabilizing protocol that guarantees that all Patricia tries converge to the unique Patricia trie that stores all publications. This means that each node that subscribes to some topic t will eventually receive all publications for t, even if the node has subscribed to the topic after some publications have already been delivered.

Chapter 9 is based on the following publication:

> M. Feldmann, C. Kolb, C. Scheideler, and T. Strothmann. "Self-Stabilizing Supervised Publish-Subscribe Systems". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, cf. [Fel+18].

We conclude the second part in Chapter 10.

List of Own Publications

Below I state a list of all publications that I co-authored during my time at Paderborn University. Some of these publications do not directly serve as a basis for this dissertation, particularly the work on discrete clock synchronization [FKS20], congestion control [FGS19] and hybrid network algorithms [FHS20]. Those publications are however by no means thematically disjoint with the topics of this dissertation. I also

co-authored a survey on algorithms for self-stabilizing overlay networks [FSS20]. The publications are listed in chronological order.

2017:

M. Feldmann and C. Scheideler. "A Self-stabilizing General De Bruijn Graph". In: *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017, cf. [FS17].

2018:

M. Feldmann, C. Kolb, C. Scheideler, and T. Strothmann. "Self-Stabilizing Supervised Publish-Subscribe Systems". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, cf. [Fel+18].

M. Feldmann, C. Scheideler and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, cf. [FSS18a].

M. Feldmann, C. Kolb, and C. 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)*, 2018, cf. [FKS18].

2019:

M. Feldmann and C. Scheideler. "Skeap & Seap: Scalable Distributed Priority Queues for Constant and Arbitrary Priorities". In: *Proceedings of the 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019, cf. [FS19].

M. Feldmann, T. Götte and C. Scheideler. "A Loosely Self-stabilizing Protocol for Randomized Congestion Control with Logarithmic Memory". In: *Proceedings of the 21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2019, cf. [FGS19].

2020:

M. Feldmann, A. Khazraei and C. Scheideler. "Time- and Space-Optimal Discrete Clock Synchronization in the Beeping Model". In: *Proceedings of the 32nd ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020, cf. [FKS20].

M. Feldmann, C. Scheideler, S. Schmid. “Survey on Algorithms for Self-Stabilizing Overlay Networks”. In: *ACM Computing Surveys*, 2020, cf. [FSS20].

M. Feldmann, K. Hinnenthal and C. Scheideler. “Fast Hybrid Network Algorithms for Shortest Paths in Sparse graphs”. In: *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS)*, 2020, cf. [FHS20].

Part I.

Distributed Data Structures

CHAPTER 2

Preliminaries

Like in sequential environments, *efficient* distributed data structures are important in order to realize efficient distributed applications. The most prominent type of distributed data structure is the distributed hash table (DHT). Many distributed data stores employ some form of DHT for lookups. Important applications include file sharing (e.g., BitTorrent), distributed file systems (e.g., PAST), publish-subscribe systems (e.g., SCRIBE), and distributed databases (e.g., Apache Cassandra). Other distributed forms of well-known data structures, such as queues, stacks, and priority queues, have been given much less attention, however, although they have a number of interesting applications as well:

- A distributed queue is used to come up with a unique ordering of messages, transactions, or jobs, and it can be used to realize fair work stealing [BL99] since tasks available in the system may be fetched in FIFO order. Other applications are distributed mutual exclusion, distributed counting, or distributed implementations of synchronization primitives. Server-based approaches of realizing a queue in a distributed system already exist, like Apache ActiveMQ, IBM MQ, or JMS queues. However, none of these implementations provide a queue that allows massively parallel accesses while keeping the amount of work for each participant balanced.
- A distributed stack is used in LIFO-based scheduling, which is useful in applications where scheduled tasks do not depend on each other. The major advantage of a stack in the distributed setting is that by *local combining* (an insertion request ins can be matched to a deletion request del if both are generated by the same node) one can potentially speed up the scheduling process in comparison to a distributed queue.
- Distributed priority queues are useful in the area of scheduling, in case the inserted tasks have been assigned priorities and in case workers preferably want to get assigned to the most prioritized tasks first. Another application for a distributed priority queue is distributed sorting.

The major problem of coming up with a fully distributed version of these data structures is that their semantics are inherently sequential. In this thesis, we present distributed protocols for queues, stacks and priority queues for a constant amount of priorities. Our protocols ensure sequential consistency, fairly distribute the communication and storage load among all members of the distributed system, and efficiently process even massive amounts of requests. By slightly weakening the semantic constraints, we present a protocol for a distributed priority queue that works for an arbitrary amount of priorities and ensures serializability. We also offer

protocols that allow to let new processes join or old processes leave the system safely without violating any semantic constraints or losing important data along the way. All our protocols work in the asynchronous message passing model and can also handle massive amounts of join and leave requests efficiently. We are not aware of any distributed data structure with comparable performance.

In general, the problem we solve in the first part of this thesis is how to provide distributed protocols that let multiple processes insert elements into and delete elements from the system. By doing so, the system behaves like a sequential data structure from the perspective of each individual process.

Outline of This Chapter. In this chapter, we introduce our model alongside some techniques that will be used by the distributed protocols for queues, stacks (Chapter 3) and priority queues (Chapter 4). We first present our model in Section 2.1. Afterwards, we define the aggregation tree as a subgraph of the *Linearized De Bruijn Graph* (LDB), which represents the network topology formed by the processes (Section 2.2). The LDB is also used to embed distributed hash tables (DHT), which we describe in Section 2.3. We finish this chapter by stating the well-known Chernoff bounds in Section 2.4, which will prove to be useful throughout the entire thesis.

2.1. Model

Network Model. We study distributed data structures consisting of multiple processes that are interconnected by an overlay network.

Definition 2.1. *The* overlay network *is given by an undirected graph $G = (V, E)$ of n nodes with the following properties:*

(a) *Each node $u \in V$ represents a single process and is identified by a unique $\mathcal{O}(\log n)$-bit identifier $id(u) \in \mathbb{N}$.*

(b) *If there is an edge $\{u, v\} \in E$, then u can send messages to v and vice versa.*

There is no global (shared) memory, so nodes perform computation exclusively on the basis of their local storage. We allow $\mathcal{O}(poly(n))$ many entries of $\mathcal{O}(\log n)$ bits for each node's local storage, so overall, each node's storage consists of $\mathcal{O}(poly(n))$ bits. Define the neighbors of node u by $N(u) = \{v \in V \mid \{u, v\} \in E\}$.

Communication Model. In order to provide a clean presentation and analysis of our algorithms, we consider a variant of the asynchronous message passing model. Nodes can be *activated* at arbitrary points in time. Upon activation of node u, u performs the following steps:

(i) Receive all messages that were sent to u after its previous activation.

(ii) Perform local computation.

(iii) Send a message to each neighbor $v \in N(u)$.

We view node activations as *atomic steps*, meaning that we do not allow activations of multiple nodes to overlap. Processing activation a_i and then going to activation a_{i+1} is considered a *step*. The order in which nodes are activated is chosen by an adversary. However, we assume that the activation of nodes is *fair*, meaning that at any point in time each node is activated after a finite amount of steps.

Definition 2.2 (Round, Computation)**.** *Let $A = (a_1, \ldots, a_N)$ be the sequence of activations for which the system is active. A subsequence $R = (a_i, \ldots, a_j)$ of A where each node is activated in R at least once is denoted a* round*. A* computation *C is a partition of A into disjoint rounds $R_1, \ldots, R_T$. We denote the* lifetime *of C by T.*

We will analyze the runtime of our algorithms by the number of rounds needed in a given computation. We assume that the system is active for a polynomial number of rounds, i.e., $T \in \mathcal{O}(poly(n))$. During the execution of our algorithms we assume that no faults occur; i.e., nodes do not crash or act maliciously and messages are neither lost nor corrupted.

An important criterion to measure the load of the system is its congestion:

Definition 2.3 (Congestion)**.** *The* congestion *of the system (under all fair activations) is the maximum number of messages that need to be processed by a single node during one activation.*

Data Structure Requests. Nodes may generate insertion and/or deletion requests on the data structure at any point in time.

An *insertion request* issued by some node u generates an $\mathcal{O}(\log n)$-bit element e from a universe $\mathcal{E}$ that has to be delivered to some node $v \in V$ responsible for e.

A *deletion request* issued by some node u demands to be matched to an element e that is currently stored in the data structure by some node $v \in V$; i.e., u has to receive e from v. The element e is identified by the semantics of the data structures, which differ for queues, stacks and priority queues (see Section 3.1 for queue- and stack-semantics and Section 4.1 for priority queue-semantics). If the data structure contains no elements, then the deletion request has to be matched to the *empty element*, denoted by $\perp$. Once a deletion request is matched to some element e, both the request and e are removed from the data structure. If a deletion request got matched to $\perp$, then just the request is removed from the data structure.

Definition 2.4 (Injection Rate)**.** *The* injection rate *of a node u is denoted by $\lambda(u) \in \mathbb{N}$ and represents the maximum number of insertion and/or deletion requests that u is able to generate in each round. Denote the* maximum injection rate *by $\Lambda = \max_{u \in V}\{\lambda(u)\}$.*

The time at which a node generates requests and the amount of generated requests is determined by an adversary, which respects the injection rate of the node. We assume that $\lambda(u) \in \mathcal{O}(poly(n))$ for each node $u \in V$ and thus $\Lambda \in \mathcal{O}(poly(n))$. Denote by m the number of elements stored by the data structure. As the system is active for at most $\mathcal{O}(poly(n))$ rounds and $\Lambda \in \mathcal{O}(poly(n))$, we have that $m \in \mathcal{O}(poly(n))$ at any point in time.

2.2. Aggregation Tree

In this section we introduce the topology that is formed by the participating nodes of the data structure. It is based on the classical d-dimensional De Bruijn graph [De 46], which we introduce in the following:

Definition 2.5 (De Bruijn Graph)**.** *Let $d \in \mathbb{N}$. The (d-dimensional)* De Bruijn graph *consists of nodes $(x_1, \ldots, x_d) \in \{0,1\}^d$ and edges $(x_1, \ldots, x_d) \to (j, x_1, \ldots, x_{d-1})$ for all $j \in \{0,1\}$.*

One can route a message via bit shifting from any source $s \in \{0,1\}^d$ to any target $t \in \{0,1\}^d$ by following exactly d edges. For example, for $d = 3$ we may route from $s = (s_1, s_2, s_3) \in \{0,1\}^3$ to $t = (t_1, t_2, t_3) \in \{0,1\}^3$ via the path $((s_1, s_2, s_3)$, (t_3, s_1, s_2), (t_2, t_3, s_1), $(t_1, t_2, t_3))$.

For our network topology we adapt a dynamic version of the De Bruijn graph, called the *Linearized De Bruijn Graph* (LDB), which is based on the distance-halving network from [NW07]:

Definition 2.6 (Linearized De Bruijn Graph)**.** *Let V be a set of n nodes with unique identifiers in $\mathbb{N}$ and let $h : \mathbb{N} \to [0,1)$ be a publicly known pseudorandom hash function. The* Linearized De Bruijn Graph (LDB) *is the graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ with the following properties:*

(a) *For each node $u \in V$, there are* 3 virtual nodes $l(u), m(u), r(u) \in \widetilde{V}$ *that are emulated by u. Denote $l(u)$ the* left virtual node *of u, $m(u)$ the* middle virtual node *of u and $r(u)$ the* right virtual node *of u. No other nodes are contained in $\widetilde{V}$.*

(b) *Each middle virtual node $m(u) \in \widetilde{V}$ has a* label *$label(m(u)) = h(id(u))$. Define the labels for $l(u)$ and $r(u)$ by $label(l(u)) = \frac{h(id(u))}{2}$ and $label(r(u)) = \frac{1+h(id(u))}{2}$, respectively. We may indistinctively use $l(u), m(u)$ or $r(u)$ to denote a virtual node or its corresponding label, when clear from the context.*

(c) *Let $(v_1, \ldots, v_{3n})$ be the total order of all virtual nodes according to their labels. $\widetilde{E}$ consists of* linear edges $\widetilde{E}_{lin}$ *and* virtual edges $\widetilde{E}_{vir}$*, where*

$$\widetilde{E}_{lin} = \{\{v_i, v_{i+1}\} \mid i \in \{1, \ldots, 3n-1\}\} \cup \{v_{3n}, v_1\}$$

and

$$\widetilde{E}_{vir} = \{\{l(u), m(u)\}, \{m(u), r(u)\} \mid u \in V\}.$$

Intuitively, the linear edges arrange all virtual nodes in a sorted ring ordered by their node labels, whereas the virtual edges can be seen as local edges between the virtual nodes of a single process u. The virtual edges are used to emulate the edges of the classical De Bruijn graph from Definition 2.5.

We say that a virtual node $u \in \widetilde{V}$ is *right* to a node $v \in \widetilde{V}$ if the label of u is greater than the label of v, meaning that $u > v$. Similarly, u is *left* to v if $u < v$. We can break ties between virtual node labels by comparing the unique identifiers of their corresponding nodes in V, as no two virtual nodes emulated by the same

node $u \in V$ can have the same label. If $u, v, w \in \widetilde{V}$ are consecutive in the linear ordering with $u < v < w$ (i.e., there exist linear edges $\{u, v\}, \{v, w\}$ in $\widetilde{G}$), we say that w is v's *successor* (denoted by $succ(v)$) and that u is v's *predecessor* (denoted by $pred(v)$). As a special case we define $pred(v_{min}) = v_{max}$ and $succ(v_{max}) = v_{min}$, where $v_{min} = v_1$ is the node with minimal label value and $v_{max} = v_{3n}$ is the node with maximal label value. This guarantees that each virtual node has a well-defined predecessor and a well-defined successor on the sorted ring. Moreover, each node u maintains two variables $pred(u)$ and $succ(u)$ for storing its predecessor and successor. We assume that u knows whether $pred(u)$ and $succ(u)$ is a left, middle or right virtual node. Otherwise, u could easily determine this within at most two rounds by directly asking $pred(u)$ if it is a left, middle or right virtual node, for example.

By adopting the result from [RSS11, Theorem 1], one can show that the LDB can emulate the routing in the classical De Bruijn graph with $d = \lceil \log n \rceil$:

Lemma 2.7. *For any $p \in [0, 1)$, routing a message from a source node $s \in \widetilde{V}$ to the node $t = \text{argmin}_{v \in \widetilde{V}, v \leq p}\{p - v\} \in \widetilde{V}$ can be done in the LDB in $\mathcal{O}(\log n)$ rounds w.h.p.*

Intuitively, this means that we can route a message from any source node s to the node t that is closest from below to p in $\mathcal{O}(\log n)$ rounds, w.h.p. In the analysis of [RSS11, Theorem 1], it is shown that the message can only fall behind by at most $\mathcal{O}(\log n)$ hops compared to its position in the d-dimensional De Bruijn graph, w.h.p. As an implication of the fact that the maximum distance between two consecutive middle virtual nodes is $\Theta(\log n/n)$ w.h.p. (see [NW07, Lemma 4.1]), it holds that a node in the LDB is responsible for the emulation of up to $\mathcal{O}(\log n)$ nodes in the d-dimensional De Bruijn graph, w.h.p. This holds due to the following argumentation: Map the label $label(u)$ of a virtual node u to the binary string $(x_1, \ldots, x_d)$ such that

$$label(u) = \sum_{i=1}^{k} x_i \cdot \frac{1}{2^i}.$$

If the distance between two consecutive middle virtual nodes is in $\Theta(\log n/n)$, then their binary strings differ only in the last $\Theta(\log \log n)$ bits, meaning that there exist at most $\mathcal{O}(2^{\log \log n}) = \mathcal{O}(\log n)$ binary strings of nodes in the classical De Bruijn graph between the two middle virtual nodes.

The following lemma summarizes the discussion above:

Lemma 2.8. *Any routing schedule with dilation[1] D and congestion C in the d-dimensional De Bruijn graph can be emulated by the LDB with $n \geq 2$ nodes with dilation $\mathcal{O}(D + \log n)$ and congestion $\widetilde{\mathcal{O}}(C)$ w.h.p.[2]*

We are now ready to define the aggregation tree as a subgraph of $\widetilde{G}$.

Definition 2.9 (Aggregation Tree)**.** *Let V be a set of n nodes and let $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ be the corresponding LDB according to Definition 2.6. The* aggregation tree $T(\widetilde{G}) = (\widetilde{V}, \widetilde{E}_T)$ *with root node v_{min} is a subgraph of $\widetilde{G}$ with the following properties:*

[1] The *dilation* of a routing schedule denotes the maximum number of hops that a packet has to travel.

[2] We use $\widetilde{\mathcal{O}}(\cdot)$ to hide polylogarithmic factors, i.e., $\widetilde{\mathcal{O}}(f(n)) = \mathcal{O}(f(n) \cdot polylog(n))$.

(a) *For each virtual node $u \in \widetilde{V}$, define the* parent $p(u)$ *of u as follows:*

- *If u is a middle virtual node, then $p(u) = l(u)$.*
- *If u is a left virtual node, then $p(u) = pred(u)$, except for the case when $u = v_{min}$ where we set $p(u) = \perp$.*
- *If u is a right virtual node, then $p(u) = m(u)$.*

(b) *For each virtual node $u \in \widetilde{V}$, define the* child nodes $C(u)$ *of u as follows:*

- *If u is a middle virtual node, then either $C(u) = \{r(u), succ(u)\}$ (if $succ(u)$ is a left virtual node) or $C(u) = \{r(u)\}$ (otherwise).*
- *If u is a left virtual node, then either $C(u) = \{m(u), succ(u)\}$ (if $succ(u)$ is a left virtual node) or $C(u) = \{m(u)\}$ (otherwise).*
- *If u is a right virtual node, then $C(u) = \emptyset$.*

The set $\widetilde{E}_T \subset \widetilde{E}$ is then defined by

$$\widetilde{E}_T = \{(p(u), u) \mid u \in \widetilde{V}, p(u) \neq \perp\} \cup \{(u, c) \mid u \in \widetilde{V}, c \in C(u)\}.$$

Observe that a virtual node can have at most 2 child nodes. Also, note that each virtual node u is able to locally detect its parent and its child nodes in the aggregation tree depending on whether u is a left, middle or right virtual node. As for each directed edge $(u, v) \in \widetilde{E}_T$, there exists a directed edge $(v, u) \in \widetilde{E}_T$, we simply view $T(\widetilde{G})$ as an undirected graph.

An example of an LDB network with 4 nodes $u, v, w, x \in V$ and its corresponding aggregation tree is given in Figure 2.1.

From Lemma 2.7, we directly obtain the upper bound for the height of the aggregation tree:

Corollary 2.10. *The aggregation tree $T(\widetilde{G})$ of $\widetilde{G}$ has the height $\mathcal{O}(\log n)$ w.h.p.*

Denote the root node v_{min} of $T(\widetilde{G})$ as the *anchor*. The aggregation tree can be used to aggregate certain values to the anchor. We call this process an *aggregation phase*. Values are *combined* with other values at each node.

Example 2.11 (Aggregation Phase)**.** *To determine the number of nodes $3n$ that participate in $T(\widetilde{G})$, each node $u \in \widetilde{V}$ initially holds the value 1. We start at the leaf nodes of $T(\widetilde{G})$, which send their value to their parent nodes upon activation. Once an inner node $v \in \widetilde{V}$ has received all values $k_1, \ldots, k_c \in \mathbb{N}$ from its c child nodes, upon activation it combines these by adding them to its own value, i.e., by computing $1 + \sum_{i=1}^{c} k_i$. Afterwards, v sends the result to its parent node $p(v)$. Once the anchor has combined the values of its child nodes with its own value it knows $3n$.*

We make heavy use of aggregation phases in our protocols. Due to Corollary 2.10 it is easy to see that an aggregation phase finishes after $\mathcal{O}(\log n)$ rounds w.h.p.

In order to ease the presentation of the algorithms that handle data structure requests, we just assume that the aggregation tree is a binary tree formed by all nodes in V that has the properties of Lemma 2.7, Lemma 2.8 and Corollary 2.10; i.e., we just neglect the fact that there is an underlying LDB network. However, in Section 3.5 we make use of the fact that the virtual nodes of the LDB network form a sorted ring, so we will reconsider the underlying LDB network specifically in this section.

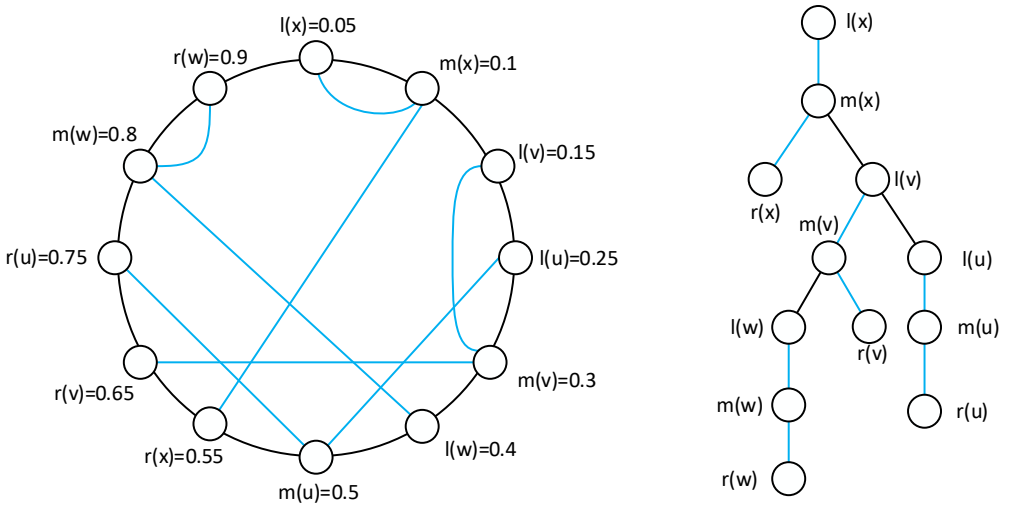


Figure 2.1.: (a) A linearized De Bruijn graph for nodes $u, v, w, x \in V$ with $m(u) = 0.5$, $m(v) = 0.3$, $m(w) = 0.8$ and $m(x) = 0.1$. Black edges represent linear edges and blue edges represent virtual edges. (b) The corresponding aggregation tree of the LDB network.

2.3. Distributed Hash Tables

In order to store the elements of the data structure in a distributed fashion, we use a distributed hash table (DHT) that makes use of consistent hashing: Elements $e \in \mathcal{E}$ that should be stored in the DHT will be assigned a unique *key* from some universe $\mathcal{K}$, denoted by $k(e) \in \mathcal{K}$. The domain of $\mathcal{K}$ may vary depending on the actual algorithm.

This key is then hashed to a real-valued position in $[0, 1)$ via a publicly known pseudorandom hash function $h : \mathcal{K} \to [0, 1)$. A virtual node u is responsible for storing all elements whose positions are within the interval $[u, succ(u))$. Thus, if we want to insert (resp. delete) an element $e \in \mathcal{E}$, we only have to search for the virtual node $u \in \widetilde{V}$ with $u \leq h(k(e)) < succ(u)$ and tell u to store e. The search for u can be performed in $\mathcal{O}(\log n)$ rounds w.h.p. according to Lemma 2.7.

Note that in the special case $u = v_{max}$, we define the the union of the two intervals $[u, 1)$ and $[0, v_{min})$ as the interval u is responsible for. So in case the label of $h(k(e))$ is strictly smaller than v_{min}, we just deliver e to v_{min} first and then delegate it to v_{max} within one additional round as $\{v_{min}, v_{max}\} \in \widetilde{E}$.

We will use the following DHT requests in our algorithms:

- Put(e, $k(e)$) Inserts the element $e \in \mathcal{E}$ with the key $k \in \mathcal{K}$ into the DHT.
- Get(k, v): Removes the element with the key $k \in \mathcal{K}$ from the DHT and delivers it to the initiator $v \in V$ of the request.

With the arguments from above, we can easily derive the following lemma on the runtime of DHT requests:

Lemma 2.12. *A Put or a Get request on the DHT can be processed by the underlying LDB in $\mathcal{O}(\log n)$ rounds w.h.p.*

It is well-known that consistent hashing is *fair*, meaning that each node stores the same number of elements for the DHT on expectation. From the fact that the expected distance between two consecutive virtual nodes is $1/3n$, we can conclude the following:

Corollary 2.13. *Let V be a set of n nodes and let $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ be the corresponding LDB according to Definition 2.6. Embedding a DHT with m elements into $\widetilde{G}$ implies that each virtual node $v \in \widetilde{V}$ stores $m/3n$ elements on expectation. Consequently, each node $u \in V$ stores m/n elements on expectation.*

Therefore, the DHT embedded into the LDB also possesses the fairness property.

Finally, we note that the nodes are able to maintain multiple distributed hash tables at once, when using different hash functions for each DHT and assuming a unique identifier for each hash table that is assigned to each element stored by a node alongside its key. Looking at each DHT individually, the statements of Lemma 2.12 and Corollary 2.13 obviously still hold.

2.4. General Notions from Probability Theory

We assume that the reader of this thesis is familiar with the concepts of random variables and their expected values. For a random variable X, we denote by $\mathbb{E}[X]$ the expected value of X. We make use of the following well-known Chernoff bounds [Che52], which we use to bound the probability for an algorithm to fail. Throughout this thesis we use the notation $\exp(x)$ instead of e^x.

Theorem 2.14 (Chernoff Bounds)**.** *Let $X_1, \ldots, X_n$ be a set of independent binary random variables. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}[X]$. Then it holds that:*

(a) *For any $0 \leq \delta \leq 1$:*

$$\Pr[X \geq (1+\delta)\mu] \leq \exp\left(\frac{-\delta^2\mu}{3}\right).$$

(b) *For any $0 \leq \delta \leq 1$:*

$$\Pr[X \leq (1-\delta)\mu] \leq \exp\left(\frac{-\delta^2\mu}{2}\right).$$

CHAPTER 3

Distributed Queues and Stacks

In this chapter we study the first two of the three mentioned fundamental distributed data structures: distributed queues and stacks. As already discussed in the previous chapter, there are some interesting applications where such distributed queues and stacks may be useful.

The first challenge one has to overcome when constructing distributed protocols for these data structures is to define precise semantic constraints that basically make the system look like a queue or a stack from the perspective of a single node. Another task is to cope with the large amount of requests potentially generated by the nodes over the data structure's lifetime. We want to use the full power of our system and spread the work to process all of these requests uniformly among all nodes such that there is no bottleneck. This also means that the elements stored in the distributed data structure should be divided onto the local storage of all nodes such that no node has to store significantly more elements than other nodes.

Another issue that specifically relates to the distributed setting is that distributed systems are dynamic by nature, meaning that the set of participants may vary over the lifetime of the system. This is why we also have to come up with distributed protocols dedicated to safely include joining nodes into the system or exclude nodes from the system that want to leave. When including new nodes into the system, we have to make sure to provide them their share of the data and also to have them work on future requests on the data structure. Excluding a node from the system has to be performed in such a manner that the node leaving the system delegates the elements it holds for the data structure to other (non-leaving) nodes.

Underlying Publications. This chapter is based on the following publication:

> M. Feldmann, C. Scheideler and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, cf. [FSS18a].

The protocol for the distributed stack (Section 3.4) is based on the full version of the paper above:

> M. Feldmann, C. Scheideler and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue", *CoRR, abs/1802.07504*, 2018, cf. [FSS18b].

Outline of This Chapter. First we introduce some basic notation and define semantic constraints specifically tailored to distributed queues and stacks (Section 3.1). We then give an overview of related work in Section 3.2 and present and analyze our

protocol for the distributed queue in Section 3.3. We proceed with our protocol for the distributed stack (Section 3.4). Finally, in Section 3.5, we investigate node dynamics; i.e., we present protocols that are able to include new nodes into the system, or exclude nodes from the system – both without violating any semantic constraints of the data structure.

3.1. Basic Notation and Semantics

Before we describe the actual protocols for distributed queues and stacks, we have to define the semantics of the insertion and deletion requests that can be invoked on the data structure by the nodes. Among other things, we make sure that the distributed queue guarantees the distributed variant of the *FIFO* property (First In – First Out) and that the distributed stack guarantees the distributed variant of the *LIFO* property (Last In – First Out).

Recall that the system consists of n nodes and $\mathcal{E}$ is the universe of all elements that may be inserted into the data structure. A protocol for the *distributed queue* has to specify the following two requests that can be generated by a node $u \in V$:

- Enqueue(e): Adds the element $e \in \mathcal{E}$ to the distributed queue.
- Dequeue(): Removes an element $e \in \mathcal{E}$ from the distributed queue such that the FIFO property is guaranteed and delivers e to u. If the queue contains no elements, $\perp$ is returned to u.

Similarly, we have the following two requests that can be invoked on the *distributed stack* by a node $u \in V$:

- Push(e): Adds the element $e \in \mathcal{E}$ to the distributed stack.
- Pop(): Removes an element $e \in \mathcal{E}$ from the distributed stack such that the LIFO property is guaranteed and delivers e to u. If the stack contains no elements, $\perp$ is returned to u.

While in standard sequential data structures it is very easy to guarantee certain properties like FIFO or LIFO, it is much harder to guarantee FIFO or LIFO in a distributed system. This is due to messages potentially being delayed or nodes not having access to a local or global clock, which are standard assumptions in the asynchronous message passing model.

For example, consider two nodes $u, v \in V$ and assume that u and v generate Enqueue requests for elements e_u and e_v at times t_u and t_v, respectively, such that $t_u < t_v$ holds. Once another node, say w, wants to take an element out of the queue via a Dequeue request, w cannot decide if according to the FIFO rule e_u or e_v has to be delivered to w as it does not know the times t_u or t_v by default. Even worse, neither u nor v can decide this for w as well, since there is no global clock that the nodes have access to. More precisely, even if both u and v may have access to a local clock and thus know the times t_u and t_v at which they generated their requests, they are not able to decide if $t_u < t_v$ holds, as the local clocks are not assumed to be perfectly in sync.

As a consequence, we want to establish a global serialization of the requests ensuring some well-defined semantics that mimic the FIFO/LIFO rules in the sequential setting. This has to be done without creating bottlenecks in the system, even when the maximum injection rate Λ is high. In order to properly define how such a global serialization looks, we introduce some notation first. For a node $u \in V$, let $\mathsf{op}(u, i)$ denote the i-th request (either an insertion or a deletion request) generated by u and denote by $\lambda_T(u) \in \mathcal{O}(T \cdot \lambda(u))$ the number of requests generated by u over the lifetime $T \in \mathcal{O}(poly(n))$ of the data structure. Further, let

$$S = \left\{ \bigcup_{i=1}^{\lambda_T(u)} \mathsf{op}(u, i) \;\middle|\; u \in V \right\}$$

be the set of all requests issued by all nodes over the lifetime of the data structure. We are now ready to provide formal definitions for some standard consistency models considered in the area of parallel and distributed computing, namely serializability, local consistency and sequential consistency:

Definition 3.1. *Let $\mathcal{DS}$ be a distributed data structure formed by a set V of nodes.*

(a) *$\mathcal{DS}$ is* serializable *if and only if there exists an ordering $\prec$ on the set S so that the distributed execution of all requests in S on the data structure is equivalent to the serial execution w.r.t. $\prec$.*

(b) *$\mathcal{DS}$ is* locally consistent *if and only if there exists an ordering $\prec$ on the set S so that for all $u \in V$ and $i \in \mathbb{N}$ it holds that $\mathsf{op}(u, i) \prec \mathsf{op}(u, i+1)$.*

(c) *$\mathcal{DS}$ is* sequentially consistent *if and only if it is serializable and locally consistent w.r.t. the same ordering $\prec$.*

A distributed data structure that is serializable provides a global ordering $\prec$ on its requests, which will be helpful to satisfy FIFO and LIFO constraints required by queues and stacks, respectively. Sequential consistency is even stricter in a sense that it satisfies local consistency in addition to serializability. Local consistency intuitively means that for each single node u, the requests issued by u have to come up in the global ordering $\prec$ in the order they were generated by u.

In order for our distributed data structure to resemble a queue or a stack, we need to introduce additional constraints. Let $\mathsf{ins}(u)$ be an insertion request (either Enqueue or Push, depending on whether we are dealing with a queue or a stack) generated by node $u \in V$. Similarly, let $\mathsf{del}(u)$ be a deletion request (either Dequeue or Pop) generated by node $u \in V$.

Definition 3.2. *A pair $(\mathsf{ins}(u), \mathsf{del}(v))$ is* matched *if $\mathsf{del}(v)$ returns the element to v that has been inserted into the data structure by u via $\mathsf{ins}(u)$. Define M to be the set of all matched requests generated over the data structure's lifetime.*

Note that not every request r has to be matched (think of an insertion request on an empty queue) and thus r is not necessarily part of some pair contained in M – we denote this by $r \notin M$.

For the distributed queue, we introduce the following queue semantics:

Definition 3.3 (Queue Semantics)**.** *Let $u, v, w, x \in V$ and let M be a matching as defined above. A protocol for the distributed queue with requests* Enqueue *and* Dequeue *is* queue consistent *if and only if there is an ordering $\prec$ on the set S such that M satisfies the following properties:*

(a) $\forall(\mathit{ins}(u), \mathit{del}(v)) \in M : \mathit{ins}(u) \prec \mathit{del}(v)$,

(b) $\forall \mathit{ins}(u), \mathit{del}(v) \notin M : \mathit{del}(v) \prec \mathit{ins}(u)$,

(c) $\forall(\mathit{ins}(u), \mathit{del}(v)), (\mathit{ins}(w), \mathit{del}(x)) \in M$ *with* $\mathit{del}(v) \prec \mathit{del}(x)$:

$$\mathit{ins}(u) \prec \mathit{ins}(w).$$

Intuitively, the three properties have the following meaning. The first property means that an element has to be inserted into the queue before it can be deleted from the queue. The second property means that a Dequeue request returning $\perp$ does so because the queue is empty at the time where the request is processed. The third property makes sure that the distributed queue behaves in a FIFO manner similar to a sequential queue.

By tweaking Definition 3.3(c) to make sure the data structure behaves in a LIFO manner, we derive the following stack semantics from Definition 3.3:

Definition 3.4 (Stack Semantics)**.** *Let $u, v, w, x \in V$ and let M be a matching as defined above. A protocol for the distributed stack with requests* Push *and* Pop *is* stack consistent *if and only if there is an ordering $\prec$ on the set S such that M satisfies the following properties:*

(a) $\forall(\mathit{ins}(u), \mathit{del}(v)) \in M : \mathit{ins}(u) \prec \mathit{del}(v)$,

(b) $\forall \mathit{ins}(u), \mathit{del}(v) \notin M : \mathit{del}(v) \prec \mathit{ins}(u)$,

(c) $\forall(\mathit{ins}(u), \mathit{del}(v)), (\mathit{ins}(w), \mathit{del}(x)) \in M$ *with* $\mathit{del}(v) \prec \mathit{del}(x)$:

$$\mathit{ins}(w) \prec \mathit{ins}(u).$$

Note that if the system consists only of a single node ($n = 1$), then the insertion and deletion requests on the distributed data structures have exactly the same semantics as in their corresponding sequential versions.

Our goal is to provide distributed protocols for queues and stacks that satisfy sequential consistency and queue/stack consistency, while still being scalable, meaning that each insertion and deletion request is processed in $\mathcal{O}(\log n)$ rounds w.h.p. We also want the overall load on the network (i.e., the number of messages that have to be sent and received) to be spread equally among all nodes, meaning that there is no bottleneck that has to deal with more messages than other nodes.

3.2. Related Work

Distributed Hash Tables. The most prominent type of data structure for distributed applications is the distributed hash table, for which the seminal work has been done by Karger et al. [Kar+97], who also introduced consistent hashing. Since then, distributed hash tables have been used by a wide range of practical applications, such as Chord [Sto+01], Pastry [RD01], Tapestry [Zha+04] or Cassandra [LM09]. Important aspects in studying distributed hash tables are, among others, routing [Bal+03; Gum+03], load balancing [Fel+14], heterogeneity [SS05] or security-related topics [SM02; DS07] concerning different attacks on a distributed hash table.

As distributed hash tables do not support range queries (that is, searching for all keys that are contained in some specific interval), distributed trees have been proposed [KW94; SSW02; SSW03; AGT10] that organize the elements of the hash table in a search tree split among all nodes.

Concurrent Queues and Stacks. There is a wealth of literature on *concurrent* data structures. In concurrent data structures, multiple processes/threads maintain and send requests to a data structure that is stored in shared memory. All of these concurrent data structures are not fully decentralized like ours, as elements of the data structure are stored in shared memory. The shared memory can be directly modified by all threads. A nice survey on many concurrent data structures (shared counters, queues, stacks, pools, linked lists, hash tables, search trees and priority queues) is written by Moir and Shavit [MS04].

As concurrent data structures share some similarities to distributed data structures, we want to give a brief overview of some related literature. Generally, one distinguishes between *blocking* and *nonblocking* implementations.

Blocking data structures have the property that a delayed thread may cause the delay of other threads as well. This usually occurs when using *locks*, which make sure that a certain part of the shared memory can only be accessed by one single thread (the one that 'holds' the lock) at any point in time. These implementations are specifically vulnerable to *memory contention*, which means that multiple threads are competing for the same location in memory with only one thread being allowed to access the location at any point in time. Consider, for example, [MS96] for blocking concurrent queues and [MS98; SZ00] for blocking concurrent stacks.

Nonblocking implementations are usually more expensive and complex as they are not allowed to use locks. Some common techniques here involve more sophisticated hardware operations like *compare-and-swap* and *load-linked/store-conditional*. Also, nonblocking data structures do not scale well when imposing strict progress conditions, which is why people came up with weaker progress conditions such as *wait-freedom* (meaning that every single operation will be completed in a finite number of steps independent from any other thread that is currently performing a requests) or *lock-freedom* (meaning that at least one of the threads currently performing an operation on the data structure completes the operation within a finite amount of time). Consider, for example, [HW90] for lock-free concurrent queues and [HSY10; GC07] for lock-free concurrent stacks. Shavit and Taubenfeld formulated some (relaxed)

semantics for wait-free concurrent queues and stacks in [ST16a].

In our distributed data structures, we use some techniques similar to the ones used in concurrent data structures: Hendler et al. [Hen+10] present a scalable synchronous concurrent queue, where they used a parallel flat-combining algorithm to combine requests (similar to the aggregation technique used by our data structure). A single 'combiner' thread gets to know requests of other threads and then executes these requests on the queue. However, they do not provide any guarantees on the semantics, meaning that it does not impose an order on the servicing of requests and is thus violating semantics like serializability and queue consistency.

Our distributed implementation for the stack makes use of local combining of Push- and Pop-requests at nodes, which is based on the *elimination* technique introduced by Shavit and Touitou [ST97].

Standard Consistency Models. *Consistency models* have been introduced in the scope of shared memory systems and define how the *output* of a system is allowed to look when getting some *input*. In our scenario, the input is represented by the insertion and deletion requests generated by the nodes, whereas the output is represented by the elements that are returned to the nodes answering their deletion requests.

There exists a large variety of consistency models, so we only briefly discuss the ones that are most relevant to this thesis. Sequential consistency (Definition 3.1(c)) was originally defined by Lamport [Lam79]. Informally speaking, it makes the whole system look like a uniprocessor system to any outsider using it. Therefore, sequential consistency is considered to be one of the strongest consistency models in multiprocessor systems. This, however, is not always advantageous, as it is generally quite costly to implement sequential consistency for a system. Note that *Linearizability* [HW90] is close, but slightly stronger than sequential consistency as it preserves real-time ordering of requests [AW94]. It is noted in [SN04] that even though linearizability is stronger, sequential consistency is the strongest consistency model used in practice.

Serializability [BSW79] (Definition 3.1(a)), although being weaker than sequential consistency, is an important criterion to check the correctness of a system, as it allows for a non-overlapping ordering of the requests. On the other hand, local consistency [BB98] (Definition 3.1(b)) refers to the weakest consistency model for shared memory as noted in [SN04]. Other consistency models that go beyond the scope of this thesis can be found in works such as [Mos93; AG96].

Distributed Queuing. Plenty of work has also been done on *distributed queuing* such as the arrow protocol [DH98; HTW01] and its follow-up works (e.g., [TH06; SB15; KW19]), but the problem considered in this research area is quite different from ours. Distributed queuing is all about the participants of the system forming a queue: Every node introduces itself to its predecessor and (depending on its position) knows its successor in the queue. It is not about inserting elements into a distributed data structure that is maintained by multiple nodes generating requests to the data structure. However, there are some important applications for distributed queuing protocols, such as coordinating the mutual exclusive access to some shared object.

3.3. Distributed Queue

Our first protocol considers the distributed queue, more specifically, the handling of Enqueue and Dequeue requests, which we will call *queue requests* in the following. The main challenge to guarantee sequential consistency (Definition 3.1(c)) lies in the fact that messages may outrun each other, since we assume an asynchronous environment with non-FIFO message delivery. In a synchronous environment this would be of no concern.

Another problem we have to solve is that the rate at which nodes issue queue requests may be very high. As long as we process each single request one by one, scalability cannot be guaranteed. Recall that the nodes form an aggregation tree into which a distributed hash table (DHT) is embedded.

3.3.1. Enqueue and Dequeue

The protocol proceeds in 4 phases. First, we aggregate batches of queue requests to the anchor of the aggregation tree. The anchor then assigns a unique key to each queue request and spreads all keys for the queue requests over the aggregation tree such that sequential consistency is fulfilled and each node receives the keys corresponding to its own generated queue request. Using those keys, nodes in the aggregation tree then generate Put and Get requests for the DHT. The following lines provide a more detailed description of this approach.

Whenever a node initiates a queue request, it stores the request in a local queue that acts as a buffer. We are going to represent a snapshot of the sequence of buffered queue requests by a *batch*:

Definition 3.5 (Batch)**.** *A* batch B *is a sequence* $(op_1, \ldots, op_k) \in \mathbb{N}_0^k$ *for which it holds that for all odd* i*,* $1 \leq i \leq k$*,* op_i *represents* op_i *many successive* Enqueue *requests. For all even* i*,* $2 \leq i \leq k$*,* op_i *represents* op_i *many successive* Dequeue *requests.*

Two batches $B_1 = (op_1, \ldots, op_k)$ and $B_2 = (op'_1, \ldots, op'_l)$ can be *combined* by computing $B = (op''_1, \ldots, op''_m)$ with $op''_i = op_i + op'_i$ and $m = \max\{k, l\}$ (set $op_i = 0$ if $i > k$ and $op'_i = 0$ if $i > l$). If a batch B is the combination of batches $B_1, \ldots, B_k$, then we denote $B_1, \ldots, B_k$ as *sub-batches* of B.

Algorithm 1 describes the phases of our protocol.

Phase 1: Aggregating Batches. At the beginning of the first phase each node u generates a snapshot of the contents of its local queue and represents it as a batch $u.B$. For example, a snapshot consisting of requests Enqueue(e_1), Enqueue(e_2), Dequeue(), Enqueue(e_3) and Dequeue() (in that specific order) is represented by the batch $(2, 1, 1, 1)$. The batch $u.B$ respects the local order in which queue requests are generated by u, which is important for guaranteeing sequential consistency.

We aggregate batches of all nodes up to the anchor via an aggregation phase (similar to Example 2.11). For this, each node u waits until it has received all batches from its child nodes $v \in C(u)$ and then combines those batches together with its own batch $u.B$ upon activation. The resulting batch, denote it by $u.B^+$, is then sent to the parent $p(u)$ of u in the aggregation tree. In addition, u memorizes the sub-batches

Algorithm 1 Handling Enqueue & Dequeue Requests in the Distributed Queue

Phase 1 (Executed at each node u)
1: Create batch $u.B$
2: Wait until u has received $v.B$ from all $v \in C(u)$
3: Combine batches $v.B$, $v \in C(u)$, with $u.B$ to a batch $u.B^+$
4: Send $u.B^+$ to $p(u)$

Phase 2 (Local computation at the anchor v_0)
5: Let $v_0.B^+ = (op_1, \ldots, op_k)$ be the combined batch from Phase 1
6: **for** $i = 1, \ldots, k$ **do**
7: **if** i is odd **then**
8: $[x_i, y_i] \leftarrow [v_0.\mathit{last} + 1, v_0.\mathit{last} + op_i]$
9: $v_0.\mathit{last} \leftarrow v_0.\mathit{last} + op_i$
10: **else**
11: $[x_i, y_i] \leftarrow [v_0.\mathit{first}, \min\{v_0.\mathit{first} + op_i - 1, v_0.\mathit{last}\}]$
12: $v_0.\mathit{first} \leftarrow \min\{v_0.\mathit{first} + op_i, v_0.\mathit{last} + 1\}$

Phase 3 (Executed at each node u)
13: Wait until u has received intervals $I = ([x_1, y_1], \ldots, [x_k, y_k])$ from $p(u)$
14: Decompose I into intervals I_u and intervals I_v for each $v \in C(u)$
15: **for all** $v \in C(u)$ **do**
16: Send I_v to v

Phase 4 (Executed at each node u)
17: Let $I_u = ([x_1, y_1], \ldots, [x_k, y_k])$
18: **for** $i = 1, \ldots, k$ **do**
19: **if** $x_i \leq y_i$ **then**
20: **if** i is odd **then**
21: Generate requests $\mathsf{Put}(e, x_i), \ldots, \mathsf{Put}(e', y_i)$
22: **else**
23: Generate requests $\mathsf{Get}(x_i, u), \ldots, \mathsf{Get}(y_i, u)$

it received from its child nodes (these sub-batches will be of use in Phase 3). At the end of the first phase, the anchor v_0 of the aggregation tree owns a batch $v_0.B^+$ that is the combination of all sub-batches $u.B$ of all nodes $u \in V$.

Phase 2: Assigning Keys. We only perform a local computation at the anchor v_0 in this phase. v_0 maintains two variables $v_0.\mathit{first} \in \mathbb{N}$ and $v_0.\mathit{last} \in \mathbb{N}_0$, such that the invariant $v_0.\mathit{first} \leq v_0.\mathit{last} + 1$ holds at any time. Upon initialization of the queue, $v_0.\mathit{first}$ is set to 1 and $v_0.\mathit{last}$ is set to 0. The interval $[v_0.\mathit{first}, v_0.\mathit{last}]$ of integers represents the keys that are currently occupied by elements of the queue, which implies that the current number of elements contained in the queue is equal to $v_0.\mathit{last} - v_0.\mathit{first} + 1$.

Now we describe how the anchor computes intervals of keys from the combined batch $v_0.B^+ = (op_1, \ldots, op_k)$ of the previous phase. On the basis of its variables $v_0.\mathit{first}$ and $v_0.\mathit{last}$, v_0 computes intervals $[x_1, y_1], \ldots, [x_k, y_k]$ by processing integers

op_i in $v_0.B^+$ in ascending order of their indices i.

If i is odd, then v_0 sets the interval $[x_i, y_i]$ to $[v_0.last+1, v_0.last+op_i]$ and increases $v_0.last$ by op_i afterwards. This indicates that op_i new elements are inserted into the queue. Similarly, if i is even, then v_0 sets the interval $[x_i, y_i]$ to $[v_0.first, \min\{v_0.first + op_i - 1, v_0.last\}]$ and updates $v_0.first$ to $\min\{v_0.first + op_i, v_0.last + 1\}$ afterwards. This indicates that op_i elements are removed from the queue. Note that in case the queue is either empty or does not hold sufficiently many elements in order to serve all op_i Dequeue requests: either $x_i = y_i + 1$ (if the queue is empty) or $y_i + 1 - x_i < op_i$ holds for the computed interval $[x_i, y_i]$.

By doing so, we implicitly assigned an interval to each sequence op_i of queue requests, and thus we can assign either a key or $\perp$ (in case some Dequeue requests are invoked on an empty queue) to each single queue request of such a sequence. We use this assignment in the next phase.

Phase 3: Decomposing Key Intervals. Once v_0 has computed all required intervals of keys $[x_1, y_1], \ldots, [x_k, y_k]$ for a batch, we start broadcasting these intervals over the aggregation tree in the following recursive manner: When a node u in the tree receives a collection $[x_1, y_1], \ldots, [x_k, y_k]$ of intervals, it decomposes these intervals with respect to each sub-batch $B_1, \ldots, B_l$ of $u.B^+$ (recall that u has memorized the composition of $u.B^+$ in Phase 1). Consider a sub-batch $B_i = (op_1, \ldots, op_m)$ of $u.B^+$. We describe how u is able to assign a (sub-)interval of keys to each op_i. Assume i is odd for op_i (corresponding to op_i many Enqueue requests). Then u assigns the (sub-)interval $[x_i, x_i + op_i - 1]$ to op_i. Afterwards, u updates $[x_i, y_i]$ by setting $[x_i, y_i] = [x_i + op_i, y_i]$. Therefore, every Enqueue request is assigned a unique key.

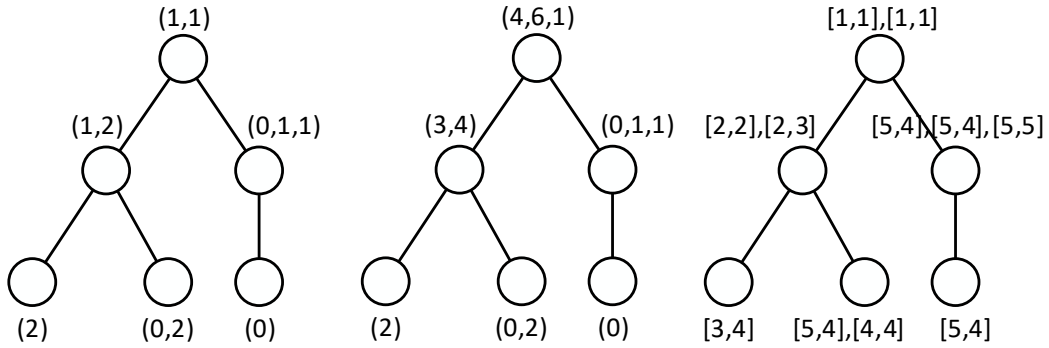
Now assume i is even for op_i (corresponding to op_i many Dequeue requests). Then u assigns the (sub-)interval $[x_i, \min\{x_i + op_i - 1, y_i\}]$ to op_i. Afterwards, u sets $[x_i, y_i] = [\min\{x_i + op_i, y_i + 1\}, y_i]$. This implies that Dequeue requests are either assigned a unique key or immediately return $\perp$ in case the interval is not large enough to assign a key to all Dequeue requests.

Once each sub-batch of $v.B^+$ has been assigned to a collection of (sub-)intervals, we send out these intervals to the corresponding child nodes in $C(v)$. Applying this procedure in a recursive manner down the aggregation tree yields an assignment of a key to all Enqueue and Dequeue requests.

Figure 3.1 illustrates an example for the first 3 phases.

Phase 4: Updating the DHT. Now that a node u knows the key $k \in \mathbb{N}_0$ for each of its queue requests, it starts generating Put and Get requests. For a request Enqueue(e) with key $k \in \mathbb{N}$, u issues a Put(e, k) request to insert e into the DHT, meaning that e is then stored at the node v responsible for the key k (see Section 2.3). This finishes the Enqueue(e) request of u.

For a Dequeue request with key k, u issues a Get(k, u) request. Since in the asynchronous message passing model it may happen that a Get(k, u) request arrives at the node v responsible for the key k in the DHT *before* the corresponding Put(e, k) request, each Get request waits at v until the corresponding Put request has arrived. This is guaranteed to happen, as we do not consider message loss. Once a node has sent out all its DHT requests, it switches back to the first phase in order to process the next batch of queue requests.



(a) After Line 1 (b) Before Phase 2 (c) After Phase 3

Figure 3.1.: Example for the computations done in the aggregation tree for the first 3 phases. The variables $u.first$ and $u.last$ of the anchor u are initially set to 1 and 0, respectively. The vectors in (a) represent the batches $u.B$ of the snapshots for all nodes u. The vectors in (b) represent the combined batches $u.B^+$ for all nodes u. The intervals in (c) represent the actual keys corresponding to the batches in (a) that nodes got assigned to their queue requests.

3.3.2. Analysis

In this section we analyze the distributed queue implemented by Algorithm 1 for its semantics, the runtime for queue requests, its congestion and the maximum size of a single message generated by the algorithm. We start with the semantics.

Lemma 3.6. *The distributed queue satisfies sequential consistency.*

Proof. We define a total order $\prec$ on all queue requests and show that $\prec$ satisfies the properties required in Definition 3.1. In order to define $\prec$, we want to assign a unique virtual value to each queue request op, indicated by $\phi(\mathsf{op}) \in \mathbb{N}$. Informally, $\phi(\mathsf{op})$ indicates the number of requests that the anchor has ever processed in Phase 2 of Algorithm 1 up to (and including) op. More formally, for a node u that generates a queue request op, let $u.B = (op_1, \ldots, op_k)$ be the batch representing the snapshot of u's local queue *right before* the creation of op. We initialize the virtual value for op as follows:

$$\phi(\mathsf{op}) = \begin{cases} op_k + 1 & \text{if } (k \text{ is odd and } \mathsf{op}{=}\mathsf{Enqueue}) \text{ or } (k \text{ is even and } \mathsf{op}{=}\mathsf{Dequeue}) \\ 1 & \text{otherwise.} \end{cases}$$

In the first case we say that op *belongs to* B *at index* k. In the second, op *belongs to* B *at index* $k+1$.

Assume for simplicity that all batches that are created in the first phase of Algorithm 1 are of the same length k. Recall that in an aggregation tree, inner nodes have at most 2 child nodes due to the way we defined the linearized De Bruijn network (Definition 2.6).

Whenever a node u combines the batches $v_1.B$, $v_2.B$ of its child nodes $v_1, v_2 \in C(u)$ with its own batch $u.B$ (Line 3), it first specifies an order $<_B$ on the batches:

i.e., it sets $u.B <_B v_1.B <_B v_2.B$. According to this order, u first combines $u.B = (op_{u,1}, \ldots, op_{u,k})$ with $v_1.B = (op_{v_1,1}, \ldots, op_{v_1,k})$ and then combines the resulting batch with $v_2.B$ in order to get the batch $u.B^+$. For a request op that belongs to $v_1.B$ at index i, we then update $\phi(\mathsf{op}) \leftarrow \phi(\mathsf{op}) + op_{u,i}$ after combining $u.B$ and $v_1.B$. Similarly, for a request op that belongs to $v_2.B$ at index i, we update $\phi(op) \leftarrow \phi(op) + op_{u,i} + op_{v_1,i}$ after combining $v_2.B$ with the already combined batches $u.B$ and $v_1.B$. Proceed in this way for every combination of batches up to the anchor.

We let the anchor maintain a (virtual) counter $c \in \mathbb{N}_0$, which initially is set to 0. Every time the anchor processes a batch $B^+ = (op_1, \ldots, op_k)$ in Phase 2, we update the value for each request op that belongs to B^+ at index i to $\phi(\mathsf{op}) \leftarrow \phi(\mathsf{op}) + c + \sum_{j=1}^{i-1} op_j$. Once we have done this for each queue request represented by the batch B^+, we update c in preparation for the next incoming batch: i.e., we set $c \leftarrow c + \sum_{i=1}^{k} op_i$.

Intuitively, imagine that the anchor processes every single queue request individually. The anchor would first consider all op_1 Enqueue requests, then all op_2 Dequeue requests, and so on. The order in which u then obtains the intervals I_u, I_{v_1} and I_{v_2} in Phase 3 is the same as in the first phase; i.e., u first obtains intervals I_u, then intervals I_{v_1} and, at last, intervals I_{v_2}.

Notice that this approach leads to each queue request getting assigned to a unique virtual value in $\mathbb{N}$. Therefore, we define $\mathsf{op}_1 \prec \mathsf{op}_2$ if and only if $\phi(\mathsf{op}_1) < \phi(\mathsf{op}_2)$ for two queue requests $\mathsf{op}_1, \mathsf{op}_2$. By definition of our protocol, $\prec$ resembles the exact order in which the anchor v_0 processes each single queue request in $v_0.B^+$. Consequently, as the key for each request is determined at the anchor in Phase 2 and the order in which the intervals are decomposed in Phase 3 is the same as the order in which batches are combined in Phase 1, the distributed execution of all requests on the queue is the same as the serial execution of all requests according to $\prec$. This implies that the distributed queue satisfies serializability (Definition 3.1(a)).

We now show that our protocol also satisfies local consistency (Definition 3.1(b)): Let $\mathsf{op}(u, i)$ be the i-th queue request and $\mathsf{op}(u, i+1)$ be the $(i+1)$-th queue request generated by node u.

If both $\mathsf{op}(u, i)$ and $\mathsf{op}(u, i+1)$ belong to the same index i in $u.B$, then $\phi(\mathsf{op}(u, i)) < \phi(\mathsf{op}(u, i+1))$ before the aggregation of batches up to the anchor starts. Once $u.B$ is combined with another batch, the virtual values of $\mathsf{op}(u, i)$ and $\mathsf{op}(u, i+1)$ always increase by the same amount (they never belong to different indices in the combined batch), so $\phi(\mathsf{op}(u, i)) < \phi(\mathsf{op}(u, i+1))$ also holds at the beginning of Phase 2.

In case $\mathsf{op}(u, i)$ and $\mathsf{op}(u, i+1)$ belong to different indices in $u.B$, say k_1 and k_2 respectively, then it holds that $k_2 = k_1 + 1$. Upon aggregating the batch $u.B$ up to the anchor in the aggregation tree, $\phi(\mathsf{op}(u, i))$ and $\phi(\mathsf{op}(u, i+1))$ are incremented by different values, such that it still holds that $\phi(\mathsf{op}(u, i)) \leq op_{k_1}$ and $\phi(\mathsf{op}(u, i+1)) \leq op_{k_1}$ for the combined batch. However, once the anchor has received the combined batch B^+, we update $\phi(\mathsf{op}(u, i)) \leftarrow \phi(\mathsf{op}(u, i)) + c + \sum_{j=1}^{k_1-1} op_j$ with

$$\phi(\mathsf{op}(u, i)) + c + \sum_{j=1}^{k_1-1} op_j \leq op_{k_1} + c + \sum_{j=1}^{k_1-1} op_j$$

and, consequently, update $\phi(\mathsf{op}(u, i+1)) \leftarrow \phi(\mathsf{op}(u, i+1)) + c + \sum_{j=1}^{k_2-1} op_j$ with

$$\begin{aligned}\phi(\mathsf{op}(u, i+1)) + c + \sum_{j=1}^{k_2-1} op_j &= \phi(\mathsf{op}(u, i+1)) + c + \sum_{j=1}^{k_1} op_j \\ &= \phi(\mathsf{op}(u, i+1)) + c + op_{k_1} + \sum_{j=1}^{k_1-1} op_j \\ &\geq \phi(\mathsf{op}(u, i+1)) + \phi(\mathsf{op}(u, i)).\end{aligned}$$

As per definition $\phi(\mathsf{op}) \geq 1$ holds for any queue request op, so it follows that $\phi(\mathsf{op}(u, i+1)) > \phi(\mathsf{op}(u, i))$.

By the definition of $\prec$, we immediately obtain $\mathsf{op}(u, i) \prec \mathsf{op}(u, i+1)$ in both scenarios, so local consistency is satisfied.

Serializability and local consistency imply that the distributed queue satisfies sequential consistency by Definition 3.1(c). □

Lemma 3.7. *The distributed queue satisfies queue consistency.*

Proof. We show that the total order $\prec$ as defined in the proof of Lemma 3.6 satisfies all conditions of Definition 3.3. Let $\mathsf{enq}(u)$ be an Enqueue request and $\mathsf{deq}(u)$ be a Dequeue request generated by node u. Let M be the set of all matched pairs of Enqueue and Dequeue requests as defined in Definition 3.2. Observe that two queue requests $\mathsf{enq}(u), \mathsf{deq}(v)$ are in M if and only if the anchor assigned the same key to them in Phase 2 of our protocol. Also, assume for convenience that the anchor v_0 processes each queue request individually in Phase 2 in the order given by $\prec$ (it is easy to see that this is equivalent to processing each $op_i \in \mathbb{N}$ of the batch $v_0.B^+$).

To show the first property (Definition 3.3(a)), assume that $(\mathsf{enq}(u), \mathsf{deq}(v)) \in M$ holds. By the above observation, it follows for the keys of these requests that $k(\mathsf{enq}(u)) = k(\mathsf{deq}(v))$. In order for the key of $\mathsf{deq}(v)$ to be equal to the key of $\mathsf{enq}(u)$, it has to hold that $k(\mathsf{deq}(v)) = v_0.\mathit{first} \leq v_0.\mathit{last}$ at the time the anchor v_0 assigns a key to $\mathsf{deq}(v)$. Therefore the key $k(\mathsf{deq}(v))$ has already been assigned to $\mathsf{enq}(u)$ beforehand, so it follows that $\mathsf{enq}(u) \prec \mathsf{deq}(v)$ and Definition 3.3(a) is satisfied.

For the second property let $\mathsf{enq}(u), \mathsf{deq}(v) \notin M$ be two arbitrary unmatched queue requests. As $\mathsf{deq}(v) \notin M$ it follows that $\mathsf{deq}(v)$ returns $\perp$ to the generator of the request. By definition of our protocol, this implies that $v_0.\mathit{first} = v_0.\mathit{last}$ at the time $\mathsf{deq}(v)$ is processed at the anchor. Therefore, it has to hold for all Enqueue requests $\mathsf{enq}(w) \prec \mathsf{deq}(v)$ (if there are any) that $k(\mathsf{enq}(w))$ has already been assigned to a Dequeue request different from $\mathsf{deq}(v)$ and therefore $\mathsf{enq}(w) \in M$. Since $\mathsf{enq}(u) \notin M$, it has to hold that $\mathsf{deq}(v) \prec \mathsf{enq}(u)$, so Definition 3.3(b) is satisfied as well.

To show Definition 3.3(c) let $(\mathsf{enq}(u), \mathsf{deq}(v)), (\mathsf{enq}(w), \mathsf{deq}(x)) \in M$ with $\mathsf{deq}(v) \prec \mathsf{deq}(x)$ and assume to the contrary that $\mathsf{enq}(u) \succ \mathsf{enq}(w)$. By definition of our protocol this implies that $k(\mathsf{enq}(u)) > k(\mathsf{enq}(w))$. However, since $(\mathsf{enq}(u), \mathsf{deq}(v))$, $(\mathsf{enq}(w), \mathsf{deq}(x)) \in M$ with $\mathsf{deq}(v) \prec \mathsf{deq}(x)$ it follows that

$$k(\mathsf{enq}(u)) = k(\mathsf{deq}(v)) < k(\mathsf{deq}(x)) = k(\mathsf{enq}(w)),$$

which implies $\mathsf{enq}(u) \prec \mathsf{enq}(w)$, which is a contradiction. □

Next, we consider the runtime of queue requests.

Lemma 3.8. *Each queue request is finished after at most $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. Consider an arbitrary queue request op generated by some node $u \in V$. By Corollary 2.10 we need $\mathcal{O}(\log n)$ rounds w.h.p. to transfer op to the anchor (Phase 1) as part of a batch. Assigning a key to op in Phase 2 only takes one single round, as it is only a local computation at the anchor. Delivering the key of op to u in Phase 3 takes $\mathcal{O}(\log n)$ rounds w.h.p., again due to Corollary 2.10. Issuing the corresponding DHT request for op takes $\mathcal{O}(\log n)$ rounds w.h.p., due to Lemma 2.12. Summing everything up, we need $\mathcal{O}(\log n)$ number of rounds w.h.p. □

We measure the congestion (Definition 2.3) of the distributed queue in the next lemma. Recall that $\Lambda \in \mathcal{O}(poly(n))$ is the maximum injection rate (Definition 2.4).

Lemma 3.9. *The distributed queue has congestion $\widetilde{\mathcal{O}}(\Lambda)$.*

Proof. At the beginning of Phase 1, each node u has at most $\lambda(u) \cdot \mathcal{O}(\log n) = \widetilde{\mathcal{O}}(\lambda(u))$ queue requests buffered, since the previous execution of Algorithm 1 lasted for $\mathcal{O}(\log n)$ rounds (Lemma 3.8) and u generates at most $\lambda(u)$ requests per round. For each of those requests, u generates a single DHT request (either Put or Get), resulting in u having to process $\widetilde{\mathcal{O}}(\lambda(u))$ requests in one activation. Since each DHT request needs $\mathcal{O}(\log n)$ rounds w.h.p. (Lemma 2.12) to finish and the aggregation tree only generates congestion up to a polylogarithmic factor (Lemma 2.8), the lemma follows. □

Finally, we analyze the message size in our distributed queue.

Lemma 3.10. *Messages generated in the distributed queue consist of $\mathcal{O}(\Lambda \log^2 n)$ bits.*

Proof. In one round each node u may generate up to Λ new queue requests. In the worst case, these requests alternate between Enqueue and Dequeue requests. Thus, the corresponding batch consists of $\mathcal{O}(\Lambda)$ bits. Due to Lemma 3.8, each node may repeat the above procedure for $\mathcal{O}(\log n)$ rounds until the previous execution of Algorithm 1 has been finished. Thus, when combining all batches of all nodes in the next execution of Algorithm 1, the resulting batch $v_0.B^+$ at the anchor v_0 consists of $\mathcal{O}(\Lambda \log^2 n)$ bits (the batch contains $\mathcal{O}(\Lambda \log n)$ entries op_i, each entry is a number in $\mathcal{O}(n)$, so it can be encoded via $\mathcal{O}(\log n)$ bits). □

The following theorem summarizes the results obtained in this section. Note that Theorem 3.11(e) is a direct implication of Corollary 2.13.

Theorem 3.11. *Algorithm 1 implements a distributed queue with the following properties:*

(a) *The distributed queue satisfies sequential consistency and queue consistency.*

(b) *Each queue request is finished after at most $\mathcal{O}(\log n)$ rounds w.h.p.*

(c) *The distributed queue has congestion $\widetilde{\mathcal{O}}(\Lambda)$.*

(d) *Messages generated in the distributed queue consist of $\mathcal{O}(\Lambda \log^2 n)$ bits.*

(e) *If the distributed queue contains m elements, each node stores m/n elements on expectation.*

3.4. Distributed Stack

In this section we show how to adapt the protocol for the distributed queue from Section 3.3 in order to work as a distributed stack with the same properties as in Theorem 3.11 (with stack consistency instead of queue consistency). Our new protocol considers Push and Pop requests, to be called *stack requests*, instead of the queue requests Enqueue and Dequeue.

3.4.1. Push and Pop

Algorithm 2 states the pseudocode for the distributed stack protocol.

Algorithm 2 Handling Push & Pop Requests in the Distributed Stack

Phase 1 (Executed at each node u)

1: Perform local combining for stack requests at each node u
2: Create batch $u.B$ ▷ $u.B = (op_1, op_2)$
3: Wait until u has received $v.B$ from all $v \in C(u)$
4: Combine batches $v.B$, $v \in C(u)$, with $u.B$ to a batch $u.B^+$
5: Send $u.B^+$ to $p(u)$

Phase 2 (Local computation at the anchor v_0)

6: Let $v_0.B^+ = (op_1, op_2)$ be the combined batch from Phase 1
7: $[x_1, y_1] \leftarrow [\max\{1, v_0.last - op_i + 1\}, v_0.last]$ ▷ Interval for Pop requests
8: $t_1 \leftarrow v_0.ticket$ ▷ Ticket for Pop requests
9: $v_0.last \leftarrow \max\{0, v_0.last - op_i\}$
10: $[x_2, y_2] \leftarrow [v_0.last + 1, v_0.last + op_2]$ ▷ Interval for Push requests
11: $[t_2, t_2'] \leftarrow [v_0.ticket + 1, v_0.ticket + op_2]$ ▷ Tickets for Push requests
12: $v_0.last \leftarrow v_0.last + op_2$
13: $v_0.ticket \leftarrow v_0.ticket + op_2$ ▷ Update *ticket* variable

Phase 3 (Executed at each node u)

14: Wait until u has received $I = \{[x_1, y_1], t_1, [x_2, y_2], [t_2, t_2']\}$ from $p(u)$
15: Decompose I into intervals I_u and intervals I_v for each $v \in C(u)$
16: **for all** $v \in C(u)$ **do**
17: Send I_v to v

Phase 4 (Executed at each node u)

18: Let $I_u = \{[x_1, y_1], t_1, [x_2, y_2], [t_2, t_2']\}$
19: **if** $x_1 \leq y_1$ **then**
20: Generate requests $\mathsf{Put}(e, (x_1, t_1)), \ldots, \mathsf{Put}(e', (y_1, t_1))$
21: **if** $x_2 \leq y_2$ **then**
22: Generate requests $\mathsf{Get}((x_2, t_2), u), \ldots, \mathsf{Get}((y_2, t_2'), u)$
23: Wait until all of the DHT requests generated above have been finished

In comparison to the distributed queue, we change the way in which the anchor computes the key intervals for Pop requests (see Phase 2 in Algorithm 2) when

processing the combined batch $v_0.B^+ = (op_1, \ldots, op_k)$. Recall that in the distributed queue the anchor v_0 computes the interval $[v_0.first, \min\{v_0.first + op_i - 1, v_0.last\}]$ in case there are op_i consecutive Dequeue requests. For op_i consecutive Pop requests, we want the anchor to return the interval $[\max\{1, v_0.last - op_i + 1\}, v_0.last]$ instead and update $v_0.last$ to $\max\{0, v_0.last - op_i\}$ afterwards. Therefore, we do not need the variable $v_0.first$ anymore. Nodes decomposing their key intervals in Phase 3 now have to take out the maximum key in the interval first.

Unfortunately, this modification does not suffice on its own in order for the stack to work correctly, because the assigned keys for inserted elements are not unique anymore: For a sequence (Push(x), Pop(), Push(y)) of requests, both Push requests will be assigned to the same key by the anchor, leading to potential conflicts in the underlying DHT. We therefore have to make sure that the key under which elements are inserted into the DHT is unique. We introduce an additional variable $v_0.ticket \in \mathbb{N}$ at the anchor, which is increased by i every time $v_0.last$ is increased by i, but is never decreased: i.e., $v_0.ticket$ is monotonically increasing. Intuitively, $v_0.ticket$ represents the number of Push requests processed at the anchor over the entire lifetime of the distributed stack, whereas $v_0.last$ represents the current size of the stack. A request is now assigned a pair $(k, t) \in \mathbb{N} \times \mathbb{N}$ instead of just a single key. For such a pair (k, t) that is assigned to a Push(x) request, we store (k, t) and x at the node that is responsible for position $h(k) \in [0, 1)$ in the DHT. A Pop request that is assigned to the pair (k, t) searches the DHT for the node u that is responsible for position $h(k) \in [0, 1)$. After arrival at u, we remove the element with ticket $t' \leq t$ from u and return it to the initiator of the Pop request. As we will see later, this element is uniquely defined.

To enhance the performance and reduce the message size of the distributed stack protocol, we let nodes *locally combine* generated stack requests in order to answer them immediately. If a node u generates k Push requests $\mathsf{push}(u, 1), \ldots, \mathsf{push}(u, k)$ followed by k Pop requests $\mathsf{pop}(u, 1), \ldots, \mathsf{pop}(u, k)$, then u can process all of these requests immediately by assigning the $(k - i + 1)$-st Push request to the i-th Pop request for all $i \in \{1, \ldots, k\}$. This is particularly advantageous in scenarios where the rate at which nodes generate requests is very high. Furthermore, it follows that all batches that are sent upwards the aggregation tree have the form $B = (op_1, op_2)$ with $op_1 \in \mathbb{N}_0$ representing Pop operations and $op_2 \in \mathbb{N}_0$ representing Push operations.

Since we consider the asynchronous message passing model, all that is left is to prevent the following scenario from happening: Consider a sequence $(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d})$ of stack requests that is processed by the anchor in Phase 2 with $\mathsf{a} = \mathsf{Push}(x)$, $\mathsf{b} = \mathsf{Pop}()$, $\mathsf{c} = \mathsf{Push}(y)$ and $\mathsf{d} = \mathsf{Pop}()$. Note that multiple executions of Phase 2 are necessary to process this sequence, as a single batch may only consist of op_1 Pop requests, followed by op_2 Push requests. Then the anchor assigns the pair (k, t) to a, (k, t) to b, $(k, t + 1)$ to c and $(k, t + 1)$ to d. Due to asynchronous behavior in our system, the DHT requests representing $\mathsf{a}, \mathsf{b}, \mathsf{c}$ and d may arrive in the order $(\mathsf{a}, \mathsf{d}, \mathsf{c}, \mathsf{b})$ at the node responsible for position $h(k)$. Thus, d returns the element x, as the ticket value t for a is smaller than the ticket value $t + 1$ for d. This leads to request b not finding an element with ticket value smaller or equal than its own, violating sequential consistency.

In order to fix this, we force all nodes u to wait in Phase 4 before switching to

Phase 1 again, until all DHT requests that u has generated in Phase 4 have been finished. This can easily be implemented by having the nodes participate in a separate aggregation phase of acknowledgments, where a leaf node sends an acknowledgment to its parent once all its DHT requests have been finished[1] and an inner node u sends an acknowledgement to its parent once all its DHT requests have been finished and it has received acknowledgments from all of its child nodes. As soon as the anchor has finished its DHT requests and received acknowledgments from all of its child nodes, we know that all DHT requests have finished. The anchor then just announces the start of the next execution of Algorithm 2 for all nodes via the aggregation tree.

Reconsidering the above example, it follows that the order of arrival of the DHT requests will be either $(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d})$ or $(\mathsf{a}, \mathsf{c}, \mathsf{b}, \mathsf{d})$, because only b and c can be together in one single batch, implying that a and d are guaranteed to be in different batches when combining requests as described above.

3.4.2. Analysis

We analyze our protocol for the distributed stack in the same manner as for the distributed queue. In this section we focus on the correctness of the semantics and the message size, as the proofs for the runtime of stack requests and the upper bound on the congestion follow directly from Lemmas 3.8 and 3.9, respectively.

Lemma 3.12. *The distributed stack satisfies sequential consistency.*

Proof. Similarly to the proof of Lemma 3.6 we specify an algorithm that assigns each stack request op a unique virtual value $\phi(\mathsf{op}) \in \mathbb{N}$ and then define $\prec$ accordingly. For the first step, we consider the stack requests that are not part of the local combining, i.e., requests that are aggregated to the anchor in Phase 1. For each of those requests op, define $\phi(\mathsf{op})$ analogously to the way we defined $\phi(\mathsf{op})$ for the queue requests in the proof of Lemma 3.6. For each node u, consider the sequence of stack requests $(\mathsf{op}(u,1), \ldots, \mathsf{op}(u,k))$ generated by u, where $\mathsf{op}(u,i)$ is the i-th stack request generated by u. We can divide the requests in such a sequence into requests that are not combined locally (and thus already got assigned to some virtual value by the first step) and pairs of requests $(\mathsf{op}(u,i), \mathsf{op}(u,i+1))$ that got combined locally.

Now we do the following: For a node $u \in V$ consider the pair $(\mathsf{op}(u,i), \mathsf{op}(u,i+1))$ of locally combined requests $\mathsf{op}(u,i)$ and $\mathsf{op}(u,i+1)$, which did not receive a virtual value yet and for which i is minimized. Let $\mathsf{op}(u,i-1)$ be u's stack request that is generated immediately before $\mathsf{op}(u,i)$. For $i \geq 2$ such a request exists and we then set $\phi(\mathsf{op}(u,i)) \leftarrow \phi(\mathsf{op}(u,i-1)) + 1$ and $\phi(\mathsf{op}(u,i+1)) \leftarrow \phi(\mathsf{op}(u,i-1)) + 2$. Afterwards, we increase the virtual values of all requests $\mathsf{op}(v,j)$ by 2 for all nodes v for which $\phi(\mathsf{op}(v,j)) \geq \phi(\mathsf{op}(u,i))$ holds. We repeat this process iteratively until there is no stack request left without a virtual value and define $\mathsf{op}_1 \prec \mathsf{op}_2$ if and only if $\phi(\mathsf{op}_1) < \phi(\mathsf{op}_2)$.

By definition of the above algorithm, each stack request receives a unique virtual value this way. Applying the same argumentation as in the proof of Lemma 3.6, we know that the distributed execution of all requests that did not get combined

[1] We have to extend the $\mathsf{Put}(e, k(e))$ request to $\mathsf{Put}(e, k(e), u)$ to make sure an acknowledgment is sent back to u after e has been received by the node responsible for the position $h(k(e))$.

locally is the same as the serial execution of these requests according to $\prec$. Observe that the pairs of locally combined requests do not influence the outcome of other stack requests and therefore serializability is satisfied. Also, it is easy to see that $\prec$ respects the local order of stack requests at each node and thus local consistency is satisfied. This finishes the proof. □

Lemma 3.13. *The distributed stack satisfies stack consistency.*

Proof. We show Definition 3.4 for the total order $\prec$ defined in the proof of Lemma 3.12. The proof works analogously to the proof of Lemma 3.7 for requests that are matched by getting assigned the same keys in Phase 2. Therefore, we need to make sure that Definitions 3.4(a) and 3.4(c) are satisfied for requests $(\mathsf{push}(u), \mathsf{pop}(v)) \in M$ that are matched due to local combining. Observe that by the definition of $\prec$ it follows that

$$\phi(\mathsf{push}(u)) < \phi(\mathsf{pop}(v)) = \phi(\mathsf{push}(u)) + 1.$$

Using this observation, Definition 3.4(a) follows trivially. Definition 3.4(c) follows as well when considering the fact that the virtual values $\phi(\mathsf{op})$ are unique after the algorithm described in the proof of Lemma 3.12 has finished. □

Lemma 3.14. *Each stack request is finished after at most $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. The proof follows from Lemma 3.8 and the fact that the number of rounds the nodes have to wait for their generated DHT-requests to finish is $\mathcal{O}(\log n)$ w.h.p. (Lemma 2.12). □

From Lemma 3.9 we directly obtain the bound on the congestion of the distributed stack.

Corollary 3.15. *The distributed stack has congestion $\widetilde{\mathcal{O}}(\Lambda)$.*

In comparison to the distributed queue protocol, we obtain a reduction in the size of a message from $\mathcal{O}(\Lambda \log^2 n)$ bits to $\mathcal{O}(\log n)$ bits:

Lemma 3.16. *Messages generated in the distributed stack consist of $\mathcal{O}(\log n)$ bits.*

Proof. Due to the local combining technique we can reduce the size of a batch in the stack to two numbers $op_1, op_2 \in \mathbb{N}_0$. As $\Lambda \in \mathcal{O}(poly(n))$ it follows that $op_1, op_2 \in \mathcal{O}(poly(n))$ as well. Therefore op_1 and op_2 can be encoded by $\mathcal{O}(\log n)$ bits. □

The following theorem summarizes our results for the distributed stack. Again, note that Theorem 3.17(e) is a direct implication of Corollary 2.13, since we spread elements pushed on the stack equally among all nodes.

Theorem 3.17. *Algorithm 2 implements a distributed stack with the following properties:*

(a) *The distributed stack satisfies sequential consistency and stack consistency.*

(b) *Each stack request is finished after at most $\mathcal{O}(\log n)$ rounds w.h.p.*

(c) *The distributed stack has congestion* $\widetilde{\mathcal{O}}(\Lambda)$.

(d) *Messages generated in the distributed stack consist of* $\mathcal{O}(\log n)$ *bits.*

(e) *If the distributed stack contains* m *elements, each node stores* m/n *elements on expectation.*

3.5. Node Dynamics

In the previous sections we assumed that the set of nodes is always static. Now we also want to support dynamic node sets, so in this section we extend the set of requests on a distributed data structure by Join and Leave requests.

- Join(): The node v issuing this request wants to join the system.
- Leave(): The node v issuing this request wants to leave the system.

When a node joins or leaves the system, this entails several changes to the system in order to get into the state assumed in the previous sections. The DHT has to be updated, which includes movement of data to joining or from leaving nodes, the LDB has to be updated and meanwhile the aggregation tree changes. To prevent chaos caused by the latter, we handle Join and Leave requests *lazily*. This means that a node v joining or leaving the network will be assigned a node u *responsible for* v. u then acts as a representative for v, meaning that u takes over v's DHT data, emulates v in the case of v being a leaving node, or relays v's insertion or deletion requests in the case of v being a joining node. Only after a sufficiently large number of nodes has requested to join or leave the system (which is counted at the anchor), the system enters a special state in which no further batches are sent out. During this state, joining nodes are fully integrated into the system (meaning they no longer need a node responsible for them) and nodes that left can end being emulated. In the following, we will specify the details of this. Keep in mind that a node that requested to join the system and that is not yet fully integrated into the system is called a *joining node* and a node that requested to leave the system and that has not yet left is called a *leaving* node.

Note that if a node v wants to join or leave the network, we have to integrate or disconnect the three virtual nodes $l(v), m(v), r(v) \in V$ into or from the system. Therefore, we generate a Join or Leave request for each of these three nodes separately. In the following we describe how each of these requests is handled. To provide a cleaner presentation, we consider these protocols to be executed by the distributed queue.

3.5.1. Join

Algorithm 3 summarizes the protocol for handling joining nodes, dividing it into a *setup phase* and an *update phase*.

Setup Phase. Assume a node v wants to join the system and further assume $v > v_0$ for now, with v_0 being the anchor (we will consider the other case separately below).

Algorithm 3 Handling Join Requests

Setup Phase (for a joining node v)

1: Delegate v to the node u with $u < v < succ(u)$ or $succ(u) < u < v$
2: Introduce u to v and move DHT data from u to v

Trigger the Start of the Update Phase

3: Aggregate #Join requests to the anchor (extend Phase 1 of Algorithm 1)
4: Start update phase once #Join requests $\geq n$

Update Phase (performed by node u, responsible for joining nodes $v_1, \ldots, v_k$)

5: $p_{old}(u) \leftarrow p(u)$
6: $C_{old}(u) \leftarrow C(u)$
7: Integrate $v_1, \ldots, v_k$ into the LDB ▷ Potentially changes $p(u)$ and/or $C(u)$
8: Wait for all acknowledgments from nodes in $C_{old}(u)$
9: Send an acknowledgments to $p_{old}(u)$
10: Wait for confirmation from $p(u)$ that the update phase has ended
11: Send confirmation that the update phase has ended to all $C(u)$

Then it sends a Join() request to a node w, where w is an arbitrary node in the system. If v wants to join the system via Join() at node w, we route v from w to the node u such that either $u < v < succ(u)$ or $succ(u) < u < v$ (in case the edge $(u, succ(u))$ closes the cycle) holds. We define u to be *responsible for* v's Join request. More precisely, u has the following tasks. First, once it has received v's reference, it introduces itself to v. Second, it hands over to v all DHT data whose positions are in v's interval. From then on, u will forward any Put or Get requests for data with positions in this interval to v. Third, u considers v to be a child in its aggregation tree, meaning that v is able to send queue requests via u. Fourth, u notifies the anchor that there is an additional node that has joined the system. For this, we extend the notion of a batch B from Definition 3.5, such that it stores an additional number $B.j \in \mathbb{N}_0$ representing the number of Join requests that the node u is responsible for.

Node u proceeds in the same manner as for queue requests. It first buffers the Join request in its local storage and once a new batch $u.B$ is created, u forwards $u.B$ up in the aggregation tree. Any intermediate node v, when combining batches $B_1, \ldots B_k$, calculates the sum of the $B_i.j$ values for the combined batch $v.B^+$. By doing so, the anchor learns a lower bound on the total number of joining nodes (note that additional nodes may have requested to join but knowledge of this has not yet reached the anchor).

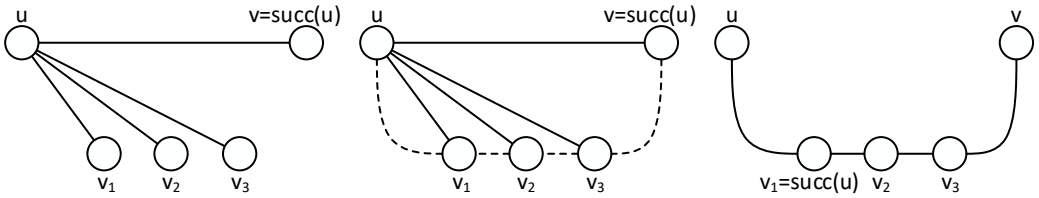
Note that a node u may become responsible for several joining nodes $v_1, \ldots v_k$. In this case, all of the above still holds with one exception: Assume u is responsible for nodes $v_1, \ldots, v_k$ and becomes responsible for an additional node v' such that a node v_i is the closest predecessor of v'. Then u does not transfer the DHT data from itself to v' but issues v_i to transfer the DHT data to v' and sends a reference of v' to v_i. Using this reference, v_i can forward any Put or Get requests that fall within the remit of v'.

Update Phase. If the anchor v_0 observes that the number of joining nodes exceeds the number of successfully integrated nodes when processing a batch, it sends the computed intervals down the aggregation tree as usual (c.f. Phase 3 of Algorithm 1), but attaches a flag to the message indicating that the *update phase* should be entered (thus informing all nodes of this). In this phase, every node will not send out a new batch until it has been informed that the update phase is over. Instead, nodes responsible for other nodes will fully integrate these nodes into the system. This works in the following way: When a node $u \neq v_0$ in the aggregation tree receives the intervals from its parent node in Phase 3, it proceeds as described in Section 3.3; i.e., it decomposes the intervals, forwards intervals to its children and possibly generates Put and Get requests in Phase 4 of Algorithm 1. Additionally, u stores its parent $p(u)$ in the aggregation tree in the variable $p_{old}(u)$ all children $C(u)$ in the variable $C_{old}(u)$. This is required because in the update phase the aggregation tree may change, but the acknowledgments that the joining nodes have been integrated successfully need to be aggregated via the old aggregation tree. That means that as soon as u has integrated all nodes it is responsible for (if any) and received acknowledgments from all nodes in $C_{old}(u)$ (if any), it sends an acknowledgment to $p_{old}(u)$ and forgets $C_{old}(u)$ and $p_{old}(u)$. The anchor v_0 behaves similarly to any other node u; i.e., it also stores its old children, processes Put and Get requests and also starts integrating nodes it is responsible for. However, when it has finished doing so and has received all acknowledgments from the nodes in $C_{old}(v_0)$, it propagates a message down in the new aggregation tree indicating that the update phase is over (recall that we consider the case of a joining node that has to become the new anchor below). This is safe because it can be shown by induction that when v_0 has received acknowledgments from all its children, every node in the tree has finished integrating at least all joining nodes that were joining when the anchor entered the update phase. Once a node has received an indication that the update phase is over, it starts aggregating and sending out batches again via Algorithm 1.

Integrating a Joining Node. We now describe how integrating a joining node works (Line 7 of Algorithm 3).

Consider a node u that is responsible for $v_1, \ldots, v_k$. W.l.o.g., we assume $u < v_1 < \ldots < v_k < succ(u)$. Node u introduces v_i to v_{i+1} and vice versa for all $i \in \{1, \ldots, k-1\}$ and introduces $succ(u)$ to v_k and vice versa. Finally, u drops its connections to $v_2, \ldots, v_k$ and $succ(u)$ and sets $succ(u)$ to v_1. Consider Figure 3.2 for an illustration of these introduction rules.

Note that the nodes v_i already stored their corresponding DHT data from the point when u became responsible for them. Due to changes in the LDB, it may happen that Put or Get requests do not need to be routed to the same target as before. However, if a $\mathsf{Put}(e, k)$ request is at a node v that is not responsible for storing the element e, v must have a neighbor that is closer to the node responsible for storing e. This is because whenever v removes an edge to a neighbor during the update phase, it has learned to know a closer one in the same direction before. Thus, v can simply forward the Put request into the correct direction. Similarly, if a $\mathsf{Get}(k)$ request is at a node v that does not store the desired element e with key k, v waits until it either stores e or it has learned to know a node that is closer to the target than itself. Since eventually our procedure forms the correct LDB topology again,



(a) Before the update phase (b) Upon integrating v_1, v_2, v_3 (c) After integrating v_1, v_2, v_3

Figure 3.2.: Introduction rules upon processing Join requests of v_1, v_2 and v_3 at node u. Dashed edges are generated by messages that u sends out to v_1, v_2, v_3 and $v = succ(u)$.

one of the above cases will eventually occur, so the Get requests will be answered.

Updating the Anchor. We now consider the special case, where at least one new node v's label is smaller than the label of the current anchor v_0. Then the node responsible for v is the node u with maximum label, i.e., $u = pred(v_0)$. In general, u behaves as described before. However, when v_0 has received all acknowledgments from its children and integrated the nodes it is responsible for, it does not send out the message indicating that the update phase is over (note that v_0 can determine that a node $v < v_0$ has joined because its neighborhood to the left has changed). Instead, v_0 first searches for the leftmost node v and then transfers its interval $[v_0.first, v_0.last]$ to v. From that point on, v will be the new anchor and send the message indicating that the update phase is over down in the new aggregation tree.

3.5.2. Leave

Algorithm 4 summarizes the protocol for handling leaving nodes, again dividing it into a setup phase and an update phase.

Setup Phase. The general strategy for handling Leave requests is the following. For each leaving node v, the node emulating the left neighbor u of v in the LDB creates a virtual node v' that acts as a replacement for v: i.e., v' will store v's DHT data, be responsible for the nodes v was responsible for and have the same connections as v had. As soon as this replacement has been created, the corresponding edges have been established, the edges to v have been removed, and all messages on their way to v have been delivered and successfully forwarded from v (we explain how to check this below), v is safe to leave the system and does so. The challenge is to deal with neighboring leaving nodes: If v has a neighbor that is also leaving, then this neighbor does not want to establish a new edge, which might result in a deadlock situation. Thus, we have to prioritize leaves. Whenever two neighboring nodes u and v determine that they both want to leave, the one with the higher identifier postpones its attempt to leave until the other one has exited the system. Since in any case there is a unique leftmost leaving node, there will always be a node that can leave the system, which inductively yields that all nodes eventually leave. To enable this, each node that calls Leave first asks all its left neighbors if it is allowed to do so. Only if all of them acknowledge does it start the actual procedure to leave. Note

Algorithm 4 Handling Leave Requests

Setup Phase (for a leaving node v)

1: Let u be the left neighbor of v in the LDB
2: u creates a virtual node v' to emulate v
3: Transfer data from v to v' via u
4: Wait until v has received all messages sent to it
5: Disconnect v from the LDB
6: v may now exit the system

Trigger the Start of the Update Phase

7: Aggregate #Leave requests to the anchor (extend Phase 1 of Algorithm 1)
8: Start update phase once #Leave requests $\geq n/2$

Update Phase (performed by node u, responsible for emulating nodes $v_1, \ldots, v_k$)

9: $p_{old}(u) \leftarrow p(u)$
10: $C_{old}(u) \leftarrow C(u)$
11: Exclude $v_1, \ldots, v_k$ from the LDB ▷ Potentially changes $p(u)$ and/or $C(u)$
12: Wait for all acknowledgments from nodes in $C_{old}(u)$
13: Send an acknowledgments to $p_{old}(u)$
14: Wait for confirmation from $p(u)$ that the update phase has ended
15: Send confirmation that the update phase has ended to all $C(u)$

that a node u that acknowledged a right neighbor v that it may leave and becomes leaving afterwards has to wait with actually executing Leave until v has left (i.e., v was successfully replaced by a virtual node v' that is now emulated by u).

One may ask how a leaving node v can determine that it has received and successfully forwarded all messages sent to it to v'. To do so, we additionally assume that for each message sent through an edge in the system, an acknowledgment is sent back to the sending node (except for acknowledgments, for obvious reasons). Each node then stores, for each edge, the number of acknowledgments it is still waiting for. Next, v asks all its neighbors to inform v once they have received all acknowledgments for messages sent to v. Once v has received all responses, it knows that it does not receive any more messages. After forwarding the received messages to v' and receiving all acknowledgments for those, it knows it is safe to leave.

A left node u that created a replacement v' for its right neighbor v is called the node *responsible for* v'. Note that v' may receive an additional Leave request from a node w. In this case, the node emulating u would spawn an additional node w' and everything is carried out as though v' were a normal node. However, we say that u is also responsible for w'. This way we make sure that only non-leaving nodes are responsible for leaving nodes. Similar to the handling of Join requests, a node u responsible for at least one leaving node sends an additional number $B.l \in \mathbb{N}_0$ in the batch $u.B$ it sends out in Phase 1 of Algorithm 1, representing the number of Leave requests that u has become responsible for since the last execution of Algorithm 1.

Update Phase. The update phase is analogous to the update phase of Algorithm 3. As soon as the number of leaving nodes becomes larger than $n/2$, the anchor

initiates the update phase during which each node u responsible for a set of virtual nodes $v_1, \ldots, v_k$ deletes these nodes and updates the LDB accordingly. Once all acknowledgments for this have been propagated up in the aggregation tree, v_0 broadcasts to all nodes in the aggregation tree that the update phase has ended. Note that both joins and leaves may be handled in the same update phase.

On a side note, one may ask what happens if a joining node v joins at some node w that is currently in the process of leaving. While w has not yet exited the system and still has edges to some non-leaving nodes, w can forward v such that v stays in the system. However, once w has exited the system and is not alive anymore, v cannot join the system through w. Still, v can detect if w is not active anymore and then try joining the system from another node.

Updating the Anchor. When v_0 wants to leave, we proceed similarly as for the join case: $pred(v_0)$ will become the node responsible for v_0 and perform the duties of the anchor. At the very end of the update phase, the anchor's information is transferred to the node v that then has the minimum identifier, making v the new anchor.

3.5.3. Analysis

We analyze the runtime of Join and Leave requests, splitting them up into the setup phase and the update phase. Note that a joining node is already allowed to generate queue requests after the corresponding setup phase has finished. Similarly, a leaving node has exited the system after its corresponding setup phase has finished; i.e., the update phase only gets rid of the virtual nodes emulating nodes that already left the system.

Theorem 3.18. *The setup phase for a joining node v is finished after $\mathcal{O}(\log n)$ rounds w.h.p.; i.e., v's Join request finishes after $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. Delegating v to the node u that is responsible for it takes $\mathcal{O}(\log n)$ rounds w.h.p. due to Lemma 2.7. Introducing u to v can be done in 2 rounds (when assuming that v acknowledges the introduction to u). □

Theorem 3.19. *Assume that v is the only node in the system issuing a Leave request. Then the setup phase for v is finished after $\mathcal{O}(1)$ rounds; i.e., v's Leave request finishes after $\mathcal{O}(1)$ rounds.*

Proof. First note that acknowledgments for each message only increase the runtime by an additional round. Creating a virtual node v' to emulate v can therefore be done in 2 rounds, as well as the transfer of data from v to v'. Waiting until v has received all messages sent to it only takes a constant amount of rounds, as all acknowledgments sent to v in a round r arrive at v no later than at the beginning of round $r + 1$. Disconnecting v from the LDB can also be done within $\mathcal{O}(1)$ rounds, because each (virtual) node in the LDB has only a constant degree and the virtual nodes form a line. □

Note that in the case where multiple (potentially connected) nodes want to leave the system, the worst-case time for such a node to be allowed to exit the system increases to $\mathcal{O}(L)$, where L is the maximum size of a connected component consisting only of leaving nodes.

Theorem 3.20. *The update phase finishes after $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. By Corollary 2.10, we need $\mathcal{O}(\log n)$ rounds w.h.p. to propagate the start of the update phase to all nodes in the aggregation tree. It is easy to see that a node u responsible for multiple joining or leaving nodes v can include/exclude them into the LDB within a constant amount of rounds at the start of the update phase. Sending out acknowledgments and broadcasting the end of the update phase can be done in $\mathcal{O}(\log n)$ rounds w.h.p., again due to Corollary 2.10. Finally, the time it takes the (old) anchor to determine the new anchor is also $\mathcal{O}(\log n)$ rounds w.h.p., as once the LDB has been updated, there it contains at most $2n$ (real) nodes and thus $6n$ virtual nodes, so searching for the virtual node with minimum position can be done in $\mathcal{O}(\log n)$ rounds w.h.p. due to Lemma 2.7. □

Having the update phase run for only $\mathcal{O}(\log n)$ rounds w.h.p. implies that it does not impact the worst-case time for insertion and deletion requests on the distributed data structure, as these also take only $\mathcal{O}(\log n)$ rounds w.h.p. to finish.

CHAPTER 4

Distributed Priority Queues and k-Selection

We continue our study of distributed data structures by looking at the distributed priority queue. As opposed to a standard queue, elements that are inserted into a priority queue come with a *priority*. The way elements are taken out of the priority queue does not depend on the order they have been inserted into it, but on the priority; i.e., we always take out the most prioritized element. This bears some additional challenges for the distributed setting, as we need a way to quickly find out at which node the most prioritized element is stored. Another factor to take into consideration will be the number of available priorities, i.e., the cardinality of the universe of available priorities.

If there is only a constant amount of priorities available, then we are able to provide a distributed protocol for a priority queue by extending the protocol for the distributed queue from the previous chapter. Simply running one distributed queue for each priority will suffice. The protocol then comes with the same properties as the protocol for the distributed queue.

If the amount of priorities is arbitrary, then we have to come up with more sophisticated techniques. To find out the set of elements that have to be removed from the priority queue when processing k deletion requests at once, we use a novel protocol that solves the *distributed k-selection problem*. The protocol runs in $\mathcal{O}(\log n)$ rounds w.h.p. and is of independent interest. Using this protocol, we are able to construct a distributed priority queue for an arbitrary amount of priorities that processes requests in $\mathcal{O}(\log n)$ rounds w.h.p. and provides serializability.

Note that the protocol for supporting Join and Leave requests from Section 3.5 can trivially be applied to both of our distributed priority queues, hence we only focus on insertion and deletion requests in this chapter.

Underlying Publication. This chapter is based on the following publication:

> M. Feldmann and C. Scheideler. "Skeap & Seap: Scalable Distributed Priority Queues for Constant and Arbitrary Priorities". In: *Proceedings of the 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019, cf. [FS19].

Outline of This Chapter. First we introduce some basic notation and define the semantics that we want our distributed priority queue to fulfill (Section 4.1). We then give an overview of related work in Section 3.2 and follow by presenting and analyzing our protocol for priority queues with a constant amount of priorities (Section 4.3). We proceed with the presentation of the protocol that solves the distributed k-selection problem (Section 4.4) and then use this protocol to obtain a distributed priority queue that works for an arbitrary amount of priorities (Section 4.5).

4.1. Basic Notation and Semantics

Similarly to Section 3.1, we start by introducing some basic notation and defining the semantics for a distributed priority queue.

As in the previous chapter, the system consists of n nodes and $\mathcal{E}$ is the universe of all elements that may be inserted into the priority queue. In addition, each element $e \in \mathcal{E}$ is assigned a unique *priority* from a universe $\mathcal{P}$. Denote e's priority by $\mathcal{P}(e)$. We allow different elements to be assigned to the same priority. Priorities in $\mathcal{P}$ can be totally ordered via $<$. Using a tiebreaker to break ties between elements having the same priority, we get a total order on all elements in $\mathcal{E}$.

A protocol for the distributed priority queue has to specify the following two requests that can be generated by a node $u \in V$:

- $\mathsf{Insert}(e)$: Adds the element $e \in \mathcal{E}$ to the distributed priority queue.
- $\mathsf{DeleteMin}()$: Removes the element $e \in \mathcal{E}$ with minimum priority from the distributed priority queue and delivers it to u. If the priority queue contains no elements, $\perp$ is returned to u.

As in the previous chapter, we want to define the semantic constraints that we will be dealing with in our implementation for the distributed priority queues. For a node $u \in V$, let $\mathsf{op}(u, i)$ denote the i-th request (either an Insert request or a DeleteMin request) generated by u and let $\lambda_T(u) \in \mathcal{O}(T \cdot \lambda(u))$ be the number of requests generated by u over the lifetime $T \in \mathcal{O}(poly(n))$ of the data structure. Let

$$S = \left\{ \bigcup_{i=1}^{\lambda_T(u)} \mathsf{op}(u, i) \;\middle|\; u \in V \right\}$$

be the set of all requests issued by all nodes over the lifetime of the data structure. We use Definition 3.1 to define serializability, local consistency and sequential consistency.

Let $\mathsf{ins}(u)$ be an Insert request generated by node $u \in V$ and let $\mathcal{P}(\mathsf{ins}(u))$ be equal to the priority of the element inserted via the $\mathsf{ins}(u)$ request. Similarly, let $\mathsf{del}(u)$ be a DeleteMin request generated by node $u \in V$. We denote a pair $(\mathsf{ins}(u), \mathsf{del}(v))$ to be *matched*, if $\mathsf{del}(v)$ returns the element to node v that was inserted into the data structure by node u via $\mathsf{ins}(u)$. Define M to be the set of all matched requests generated over the data structure's lifetime. Note that not every request r has to be matched and, thus, not every r is necessarily part of some pair contained in M – we denote this by $r \notin M$.

For the distributed priority queue, we introduce the following queue semantics, similar to Definitions 3.3 and 3.4.

Definition 4.1 (Priority Queue Semantics)**.** *Let $u, v, w, x \in V$ and let M be a matching as defined above. A protocol for the distributed priority queue with requests Insert and DeleteMin is* priority queue consistent *if and only if there is an ordering $\prec$ on the set S such that M satisfies the following properties:*

(a) $\forall(\mathsf{ins}(u), \mathsf{del}(v)) \in M : \mathsf{ins}(u) \prec \mathsf{del}(v)$,

(b) $\forall \mathit{ins}(u), \mathit{del}(v) \notin M : \mathit{del}(v) \prec \mathit{ins}(u)$.

(c) $\forall(\mathit{ins}(u), \mathit{del}(v)) \in M$ *there is no* $\mathit{ins}(w) \notin M$ *such that*

$$\mathit{ins}(w) \prec \mathit{del}(v) \ \wedge \ \mathcal{P}(\mathit{ins}(w)) < \mathcal{P}(\mathit{ins}(u)).$$

Intuitively, the three properties have the following meaning. The first property means that an element has to be inserted into the priority queue before it can be deleted. The second property means that a DeleteMin request returning $\perp$ does so because the priority queue is empty at the time where the request is processed. The third property makes sure that the distributed priority queue removes elements in a manner similar to a sequential priority queue: i.e., out of all elements that are contained in the queue, we remove the element with minimum priority. Note that Definition 4.1(c) can be modified such that the priority queue takes out the element with maximum priority instead, hence supporting DeleteMax requests instead of DeleteMin requests.

4.2. Related Work

As we already have presented the most important related work for distributed data structures in Section 3.2, we only mention additional related work specifically relevant to priority queues in this section as well as related work on distributed k-selection.

Priority Queues. Many priority queue algorithms can be found in the area of parallel computing. They mostly revolve around organizing the elements in a certain topology, for example in heaps [Aya90; Hun+96], B^+-trees [Joh94], or skip lists [ST05].

In [SZ99], the authors focus on a fixed range of priorities and come up with a technique that is based on *combining trees* [Got+98; GVW89], which are similar to the aggregation tree in our work. However, there still is a bottleneck, as the node that is responsible for a combined set of operations has to process them all by itself on the shared memory. The authors of [SL00] propose a concurrent priority queue for an arbitrary amount of priorities, where elements are sorted in a skip list. Their data structure satisfies linearizability but the realization of DeleteMin generates memory contention, as multiple nodes may compete for the same smallest element with only one node being allowed to actually delete it from the priority queue.

Maintaining the elements in one of the above topologies can be costly for the nodes in our distributed system, as this would mean that additional data have to be stored for each element stored by a process. For example, when embedding a skip list into our system, a node storing an element e of this skip list also has to store references to the nodes that store the neighbors of e. This comes at the cost of storage overhead and the problem that the skip list has to be updated every time a batch of priority queue requests has been processed, which is generally more costly than just storing the elements in a DHT.

A scalable distributed priority queue called *SHELL* has been presented by Scheideler and Schmid in [SS09]. *SHELL*'s topology resembles the De Bruijn graph and is shown to be very resilient against Sybil attacks. However, *SHELL* is concerned with

the participants of the system forming a heap and not a distributed data structure that maintains elements (cf. the paragraph on distributed queuing in Section 3.2).

Distributed k-Selection. k-selection is a classical problem that has been studied for various settings, see for example [KP91; Man+93; KNR96; RS97]. The problem has also been studied in the distributed setting for various types of data structures like cliques [RSS86], rings, meshes or binary trees [Fre83].

Kuhn et al. [KLW07] showed a lower bound of $\Omega(D \log_D n)$ on the runtime for any *generic* distributed selection algorithm, where D is the diameter of the network topology. By 'generic' they mean that the only purpose to access an element is for comparison. However, they assume the network topology to be static, which does not hold for our protocol, since we are allowed to create additional temporary edges by forwarding node identifiers via messages. This comes with the advantage that the runtime of our algorithm is only logarithmic in the number of nodes n.

Haeupler et al. [HMS18] came up with an algorithm that solves the distributed k-selection problem in $\mathcal{O}(\log n)$ rounds w.h.p. in the uniform gossip model using $\mathcal{O}(\log n)$-bit messages. This matches our result for distributed k-selection in both time and message complexity. The idea of their algorithm is to compute an approximation for the k-th smallest element through sampling and then use this algorithm several times to come up with an exact solution. While our algorithm for distributed k-selection shares some ideas regarding the sampling technique, we are able to find the k-th smallest element among $m = poly(n)$ elements distributed over n nodes, whereas the algorithm from [HMS18] works only on n elements to the best of our knowledge.

4.3. Constant Priorities

The first protocol for a distributed priority queue works for a constant amount of priorities. Formally, this means that we assume that $\mathcal{P} = \{1, \ldots, c\}$ for a constant $c \in \mathbb{N}$. The actual protocol is an extension of our protocol for distributed queues (Section 3.3) and is able to achieve the same runtimes for its requests as well as satisfying sequential consistency and priority queue consistency. Throughout the rest of this chapter, a *priority queue request*, or simply *request*, is either an Insert or a DeleteMin request.

4.3.1. Insert and DeleteMin

The idea of the protocol is to maintain a distributed queue for each priority $p \in \mathcal{P}$. As the number of priorities is only a constant, the overall overhead is a constant as well. A DeleteMin request is then processed on the non-empty queue with minimal priority. In order to support individual queues for the priorities, we extend the notion of a batch from Definition 3.5.

Definition 4.2 (Priority Queue Batch)**.** *A* priority queue batch *is a sequence* $(ins_1, del_2, \ldots, ins_{2k-1}, del_{2k})$*, where each* ins_i *is a sequence* $(ins_{i,1}, \ldots, ins_{i,|\mathcal{P}|}) \in \mathbb{N}_0^{|\mathcal{P}|}$ *representing the number of* Insert *requests for elements with priority* $p \in \mathcal{P}$ *and each* $del_i \in \mathbb{N}_0$ *represents the length of the* i*-th* DeleteMin *sequence.*

We will simply refer to a priority queue batch as a *batch* in the following. Two batches $B_1 = (ins_1, del_2, \ldots, ins_{2k-1}, del_{2k})$ and $B_2 = (ins'_1, del'_2, \ldots, ins'_{2k-1}, del'_{2k})$ can be *combined* by computing

$$B = (ins_1 + ins'_1, del_2 + del'_2, \ldots, ins_{2k-1} + ins'_{2k-1}, del_{2k} + del'_{2k})$$

where $ins_1 + ins'_1 = (ins_{i,1} + ins'_{i,1}, \ldots, ins_{i,|\mathcal{P}|} + ins'_{i,|\mathcal{P}|})$.

Algorithm 5 describes the phases of our protocol. The protocol consists of 4 phases that work similarly to the phases in the distributed queue (Algorithm 1), so in the following we only describe the points at which we have to make changes.

In the first phase, the nodes have to take the priorities of the elements that should be inserted into account when creating the snapshot of its local buffer and representing it as a batch. For example, a snapshot consisting of requests Insert(e_1), Insert(e_2), DeleteMin(), Insert(e_3) and DeleteMin() (in that specific order) with $\mathcal{P}(e_1) = 1$, $\mathcal{P}(e_2) = 1$ and $\mathcal{P}(e_3) = 2$ is represented by the batch $((2,0),1,(0,1),1)$. By doing so, the batch $u.B$ respects the local order in which requests are generated by u, which is important for guaranteeing sequential consistency.

In the second phase, we extend the set of variables that are maintained at the anchor v_0: Instead of two variables $v_0.first \in \mathbb{N}$, $v_0.last \in \mathbb{N}_0$, v_0 now maintains variables $v_0.first_p \in \mathbb{N}$, $v_0.last_p \in \mathbb{N}_0$ for each priority $p \in \mathcal{P}$. Upon initialization of the queue, each $v_0.first_p$ is set to 1 and $v_0.last_p$ is set to 0. The pair $(v_0.first_p, v_0.last_p)$ is representing the keys that are currently occupied by elements with priority p. Consequently, when the anchor processes a sequence $ins_i = (ins_{i,1}, \ldots, ins_{i,|\mathcal{P}|})$ of Insert requests, it processes each (sub-)sequence $ins_{i,p}$ using the pair $(v_0.first_p, v_0.last_p)$ in the same manner as shown before on the distributed queue. When processing a sequence del_i of DeleteMin requests, the anchor first looks for the minimum priority $p \in \mathcal{P}$ for which the corresponding queue is non-empty, i.e., for which $v_0.first_p \leq v_0.last_p$ holds. It then removes up to del_i elements from this queue in the same manner as in Phase 2 of Algorithm 1. If the queue contains less than del_i elements, say $d < del_i$, all d elements are removed from the queue and the anchor proceeds removing the remaining $del_i - d$ elements from the next non-empty queue with minimum priority until del_i elements have overall been removed, or all of the queues are empty.

In the third phase, the intervals I that have to be decomposed are now of the form $I = (I_1, \ldots, I_{2k})$ with $I_i = ([x_{i,1}, y_{i,1}], \ldots, [x_{i,|\mathcal{P}|}, y_{i,|\mathcal{P}|}])$, which basically represents the keys the anchor assigned to all Insert requests for all priorities and all DeleteMin requests. The universe $\mathcal{K}$ of keys now contains pairs (k, p), with $k \in \mathbb{N}$ and $p \in \mathcal{P}$ instead of just single numbers, so we can avoid the case where the anchor hands out the same key for two requests of different priorities.

Finally, in Phase 4 of Algorithm 5 the nodes generate the DHT requests in the same manner as in Phase 4 of Algorithm 1, with the exception that they now have to consider each of the priorities individually. Once a node has sent out all its DHT requests, it switches back to the first phase in order to process the next batch of queue requests.

Algorithm 5 Insert & DeleteMin Requests in the Distributed Priority Queue

Phase 1 (Executed at each node u)
1: Create batch $u.B$
2: Wait until u has received $v.B$ from all $v \in C(u)$
3: Combine batches $v.B$, $v \in C(u)$, with $u.B$ to a batch $u.B^+$
4: Send $u.B^+$ to $p(u)$

Phase 2 (Local computation at the anchor)
5: Let $v_0.B^+ = (ins_1, del_2, \ldots, ins_{2k-1}, del_{2k})$ be the combined batch
6: **for** $i = 1, \ldots, 2k$ **do**
7: **if** i is odd **then**
8: Let $ins_i = (ins_{i,1}, \ldots, ins_{i,|\mathcal{P}|})$
9: **for** $p \in \{1, \ldots, |\mathcal{P}|\}$ **do**
10: $[x_{i,p}, y_{i,p}] \leftarrow [v_0.last_p + 1, v_0.last_p + ins_{i,p}]$
11: $v_0.last_p \leftarrow v_0.last_p + ins_{i,p}$
12: **else**
13: $d \leftarrow del_i$ ▷ Remaining number of elements to remove
14: **while** $d > 0 \ \wedge \ \exists p \in \mathcal{P} : v_0.first_p \leq v_0.last_p$ **do**
15: Let $p \in \mathcal{P}$ be the minimum priority such that $v_0.first_p \leq v_0.last_p$
16: $[x_{i,p}, y_{i,p}] \leftarrow [v_0.first_p, \min\{v_0.first_p + d - 1, v_0.last_p\}]$
17: $v_0.first_p \leftarrow \min\{v_0.first_p + d, v_0.last_p + 1\}$
18: $d \leftarrow d - (y_{i,p} - x_{i,p} + 1)$

Phase 3 (Executed at each node u)
19: Wait until u has received intervals I from $p(u)$
20: Decompose I into intervals I_u and intervals I_v for each $v \in C(u)$
21: **for all** $v \in C(u)$ **do**
22: Send I_v to v

Phase 4 (Executed at each node u)
23: Let $I_u = (I_1, \ldots, I_{2k})$ with $I_i = ([x_{i,1}, y_{i,1}], \ldots, [x_{i,|\mathcal{P}|}, y_{i,|\mathcal{P}|}])$
24: **for** $i = 1, \ldots, k$ **do**
25: **if** i is odd **then**
26: **for** $p = 1, \ldots, |\mathcal{P}|$ **do**
27: **if** $x_{i,p} \leq y_{i,p}$ **then**
28: Generate requests $\mathsf{Put}(e, (x_{i,p}, p)), \ldots, \mathsf{Put}(e', (y_{i,p}, p))$
29: **else**
30: **for** $p = 1, \ldots, |\mathcal{P}|$ **do**
31: **if** $x_{i,p} \leq y_{i,p}$ **then**
32: Generate requests $\mathsf{Get}((x_{i,p}, p), u), \ldots, \mathsf{Get}((y_{i,p}, p), u)$

4.3.2. Analysis

We analyze our protocol for the same properties as we did for the distributed queue and the distributed stack in the previous chapter. The proofs for the runtime of

requests and the upper bound on the congestion follow directly from Lemmas 3.8 and 3.9, respectively.

Lemma 4.3. *The distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(1)$ satisfies sequential consistency.*

Proof. For a batch $B = (ins_1, del_2, \ldots, ins_{2k-1}, del_{2k})$ with $ins_i = (ins_{i,1}, \ldots, ins_{i,|\mathcal{P}|})$ replace ins_i by setting $ins_i = \sum_{j=1}^{|\mathcal{P}|} ins_{i,j}$. Then apply the same algorithm as described in the proof of Lemma 3.12 in order to assign a unique virtual value $\phi(\mathsf{op})$ to each priority queue request op.

The rest of the proof is analogous to the proof of Lemma 3.12, except for one minor point: For two Insert requests $\mathsf{ins}(u, i)$, $\mathsf{ins}(u, i+1)$ generated by the same node u with $\mathsf{ins}(u, i) \prec \mathsf{ins}(u, i+1)$, it does not necessarily hold that both requests are processed in the order induced by $\prec$ at the anchor in Phase 2 of our protocol, in case they have different priorities. One could assume that this would violate serializability. However, this only holds for Insert requests that are represented by the same value ins_i of the batch $u.B$, so there is no DeleteMin request del with $\mathsf{ins}(u, i) \prec \mathsf{del} \prec \mathsf{ins}(u, i+1)$. The distributed execution of all priority queue requests is therefore still equivalent to the serial execution w.r.t. $\prec$. □

Lemma 4.4. *The distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(1)$ satisfies priority queue consistency.*

Proof. The properties defined in Definitions 4.1(a) and 4.1(b) follow from the proof of Lemma 3.7. For the third property, note that the anchor always removes elements first from the queue with minimum priority that is non-empty. Therefore, there cannot exist an Insert request $\mathsf{ins}(w)$ that has been processed by the anchor before some DeleteMin request $\mathsf{del}(v)$ that got matched to an Insert request $\mathsf{ins}(u)$ with $\mathcal{P}(\mathsf{ins}(w)) < \mathcal{P}(\mathsf{ins}(u))$. Hence Definition 4.1(c) is satisfied as well. □

Lemma 4.5. *Messages generated in the distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(1)$ consist of $\mathcal{O}(\Lambda \log^2 n)$ bits.*

Proof. The proof follows directly from Lemma 3.10 when keeping the fact in mind that the number of priorities $|\mathcal{P}|$ is only constant and thus the number of bits by which a batch can be stored only multiplies by a constant factor, in comparison to batches in the distributed queue. □

The following theorem summarizes the results obtained in this section. Again, note that Theorem 4.6(e) is a direct implication of Corollary 2.13.

Theorem 4.6. *Algorithm 5 implements a distributed priority queue for a constant amount of priorities with the following properties:*

(a) *The distributed priority queue satisfies sequential consistency and priority queue consistency.*

(b) *Each priority queue request is finished after at most $\mathcal{O}(\log n)$ rounds w.h.p.*

(c) *The distributed priority queue has congestion $\widetilde{\mathcal{O}}(\Lambda)$.*

(d) *Messages generated in the distributed priority queue consist of* $\mathcal{O}(\Lambda \log^2 n)$ *bits.*

(e) *If the distributed priority queue contains* m *elements, each node stores* m/n *elements on expectation.*

4.4. Distributed k-Selection

In this section we present a novel protocol that solves the distributed k-selection problem in $\mathcal{O}(\log n)$ rounds. We will use this protocol in Section 4.5 in order to construct a priority queue with arbitrarily many priorities. The protocol that we propose in this section might be of independent interest. Throughout this section we are given an aggregation tree of n nodes with $m \in \mathbb{N}$ elements distributed uniformly among all nodes; i.e., each node u stores m/n elements on expectation. Denote by $u.E$ the set of elements stored at node u. Recall that the storage capacity of each node is polynomial in n, so $m \in \mathcal{O}(poly(n))$: i.e., $m \leq n^q$ for a constant $q \in \mathbb{N}$. Consider the ordering $e_1 < \ldots < e_m$ of all elements stored in the priority queue according to their priorities $\mathcal{P}(e_i)$. We denote the *rank* of an element e_i in this ordering by $rank(e_i) = i$. As we will use distributed k-selection for a priority queue with arbitrarily many priorities, we assume that the set of priorities is larger than some constant now, i.e., $\mathcal{P} = \{1, \ldots, n^q\}$. Also, we may assume for convenience that each element has a unique priority, otherwise one could just use a tiebreaker to determine the rank of two elements with the same priority.

Definition 4.7. *Given a value* $k \in \mathbb{N}$*, the* distributed k-selection problem *is the problem of determining the* k*-th smallest element out of a set of* $m = \mathcal{O}(poly(n))$ *elements, i.e., the element* $e \in \mathcal{E}$ *with* $rank(e) = k$*.*

For scalability reasons we allow nodes to send messages of at most $\mathcal{O}(\log n)$ bits only.

Each node u maintains a set $u.C \subseteq u.E$ that represents the remaining *candidates* for the k-th smallest element at u. Denote the set of all candidates by $C = \bigcup_{u \in V} u.C$ and the number of remaining candidates by $N = |C|$. Initially each node u sets $u.C$ to $u.E$, which leads to $N = m$. We assume that the anchor initially knows the values n and m (and thus also knows an appropriate value for q) as these can easily be computed via a single aggregation phase. The anchor v_0 keeps track of values N and k throughout all phases of our protocol via variables $v_0.N$ and $v_0.k$. Note that once we are able to reduce N, we also have to update the value for k, because removing a single candidate with a rank less than k implies that we only have to search for the $(k-1)$-th smallest element for the remaining candidates.

We dedicate this section to the proof of the following theorem:

Theorem 4.8. *There exists a protocol that solves the distributed* k*-selection problem in* $\mathcal{O}(\log n)$ *rounds and congestion* $\widetilde{\mathcal{O}}(1)$ *w.h.p., using* $\mathcal{O}(\log n)$*-bit messages.*

The protocol (Algorithm 6) works in three phases. In the first phase we perform a series of $\log(q) + 1$ aggregation phases in order to reduce the number of possible candidates from n^q to $\mathcal{O}(n^{3/2} \cdot \log n)$ elements. The second phase further reduces this number to $\mathcal{O}(\sqrt{n})$ candidates via aggregating $\sqrt{n}$ sample elements in parallel.

In the last phase we directly compute the k-th smallest element out of the remaining $\mathcal{O}(\sqrt{n})$ candidates.

Algorithm 6 Protocol for Distributed k-Selection

Input: n, $m = n^q$, k
Output: $e_k \in \mathcal{E}$ with $rank(e_k) = k$

Initialization
1: $v_0.N \leftarrow m$
2: $v_0.k \leftarrow k$

Phase 1 (Repeat $\log(q) + 1$ times)
3: Propagate n, $v_0.k$ to all nodes
4: Compute $u.P_{min}, u.P_{max} \in \mathcal{P}$ at each node $u \in V$
5: Compute $P_{min} = \min_{u \in V}\{u.P_{min}\}$ and $P_{max} = \max_{u \in V}\{u.P_{max}\}$
6: Remove candidates with priorities not in $[P_{min}, P_{max}]$
7: Update $v_0.k$, $v_0.N$

Phase 2 (Repeat until $v_0.N \leq \sqrt{n}$)
8: Propagate n, $v_0.N$ to all nodes
9: For each $e \in C$: Include e into C' with probability $\sqrt{n}/N$
10: Sort candidates $e_1, \ldots, e_{n'} \in C'$ based on their priority
11: Fix $\zeta \in \Theta(\sqrt{\log n} \cdot \sqrt[4]{n})$
12: Determine $e_l, e_r \in C'$ with $l = \lfloor k\frac{n'}{N} - \zeta \rfloor$ and $r = \lceil k\frac{n'}{N} + \zeta \rceil$
13: Remove candidates with priorities not in $[\mathcal{P}(e_l), \mathcal{P}(e_r)]$
14: Update $v_0.k$, $v_0.N$

Phase 3
15: Sort remaining candidates based on their priority
16: **return** e_k

4.4.1. Phase 1: Sampling

The first phase involves $\log(q)+1$ iterations. At the start of each iteration, the anchor propagates the values of k and n to all nodes via an aggregation phase. Then each node u computes the priorities of the $\lfloor k/n \rfloor$-th and the $\lceil k/n \rceil$-th smallest candidates of $u.C$. Let these priorities be denoted by $u.P_{min}$ and $u.P_{max}$. The nodes then aggregate these priorities up to the anchor, such that in the end the anchor receives priorities $P_{min} = \min_{u \in V}\{u.P_{min}\}$ and $P_{max} = \max_{u \in V}\{u.P_{max}\}$. The anchor then instructs all nodes u in the aggregation tree to remove all candidates from $u.C$ with a priority less than P_{min} or larger than P_{max}. Afterwards, the nodes aggregate the number of candidates removed this way up to the anchor. Let k_{min} denote the number of removed candidates with a priority less than P_{min} and let k_{max} be the number of removed candidates with a priority larger than P_{max}. The anchor updates $v_0.N$ and $v_0.k$ by setting $v_0.N \leftarrow v_0.N - (k_{min} + k_{max})$ and $v_0.k \leftarrow v_0.k - k_{min}$.

We obtain the following two lemmas which basically show that once the first phase has finished we are left with $\mathcal{O}(n^{3/2} \cdot \log n)$ candidates, one of them being the k-th smallest element.

Lemma 4.9. *Let $e_k \in C$ be the element with rank k. Then $P_{min} \leq \mathcal{P}(e_k) \leq P_{max}$.*

Proof. We first show $P_{min} \leq \mathcal{P}(e_k)$. Assume to the contrary that $P_{min} > \mathcal{P}(e_k)$. Then each node $u \in V$ has chosen $u.P_{min}$ with $u.P_{min} > \mathcal{P}(e_k)$. Thus, the $\lfloor k/n \rfloor$-th element of any node u has a priority strictly larger than $\mathcal{P}(e_k)$. It follows that u has stored at most $\lfloor k/n \rfloor - 1$ elements with priorities at most $\mathcal{P}(e_k)$. This implies that the number of elements with rank less that or equal to k is at most

$$\begin{aligned}(\lfloor k/n \rfloor - 1) \cdot n &\leq (k/n - 1) \cdot n \\ &= k - n \\ &\leq k - 1,\end{aligned}$$

which is a contradiction.

Now assume to the contrary that $P_{max} < \mathcal{P}(e_k)$. Then each node $u \in V$ has chosen $u.P_{max}$ with $u.P_{max} < \mathcal{P}(e_k)$. Thus, the $\lceil k/n \rceil$-th element of any node u has a priority strictly less than $\mathcal{P}(e_k)$. It follows that u has stored at most $|u.C| - \lceil k/n \rceil$ elements with priorities greater than or equal to $\mathcal{P}(e_k)$. This implies that the number of elements with rank greater than or equal to k is at most

$$\begin{aligned}\sum_{u \in V} (|u.C| - \lceil k/n \rceil) &= \left(\sum_{u \in V} |u.C| \right) - n \cdot \lceil k/n \rceil \\ &= N - n \cdot \lceil k/n \rceil \\ &\leq N - n \cdot k/n \\ &= N - k,\end{aligned}$$

which is a contradiction. □

Lemma 4.10. *After $\log(q) + 1$ iterations of Phase 1, $N \in \mathcal{O}(n^{3/2} \cdot \log n)$ w.h.p.*

Proof. First we want to compute how many candidates are left in variables $u.C$ after a single iteration of Phase 1. Let X_i be the event that the candidate c_i with $rank(c_i) = i$ is stored at node u for a fixed $u \in V$. Then $\Pr[X_i = 1] = 1/n$. Let $X = \sum_{i=1}^{k} X_i$. Then $\mathbb{E}[X] = k/n$. X denotes the number of candidates stored at u with rank within $[1, k]$. We show that the rank of the $\lfloor k/n \rfloor$-th smallest candidate in $u.C$ deviates from k by only $\mathcal{O}(\sqrt{nk \log n})$ w.h.p. When using Chernoff bounds (Theorem 2.14(b)) we get that

$$\Pr\left[X \leq (1 - \varepsilon)\frac{k}{n}\right] \leq \exp\left(-\varepsilon^2 \frac{k}{2n}\right) \leq n^{-c}$$

for $\varepsilon = \sqrt{(c \log n) \cdot 2n/k}$ and a constant c. So with high probability, each node u has at least $(1 - \varepsilon) \cdot \frac{k}{n}$ candidates with rank within $[1, k]$ stored in $u.C$. It follows that the rank of the $\lfloor k/n \rfloor$-th smallest candidate chosen by u is at least $(1 - \varepsilon) \cdot k$ w.h.p.

By the union bound we know that w.h.p. the rank of the candidate with priority P_{min} is at least $(1-\varepsilon)\cdot k$, so it deviates from k by at most

$$k \cdot \varepsilon = k \cdot \sqrt{(c \log n) \cdot 2n/k} = \mathcal{O}(\sqrt{nk \log n}).$$

We want to remark that we are only allowed to apply the Chernoff bounds (Theorem 2.14(b)) in case $k \geq c \log n \cdot 2n$. In case that $k < c \log n \cdot 2n$, the rank of the candidate with priority P_{min} deviates from k by at most $k - 1 < c \log n \cdot 2n \in \mathcal{O}(n^{3/2} \cdot \log n)$.

Using Theorem 2.14(a), we can analogously show that the rank of the candidate with priority P_{max} deviates from k by no more than $\mathcal{O}(\sqrt{nk \log n})$ w.h.p.

So the number of candidates left after the first iteration of Phase 1 is at most $\mathcal{O}(\sqrt{nk \log n})$ w.h.p. As we perform Phase 1 on the remaining candidates recursively for $\log(q) + 1$ iterations, we get the following function T for the number of remaining candidates after $i \geq 1$ iterations of Phase 1:

$$T^i(k) = \left(\prod_{j=1}^{i} \sqrt[2^j]{n} \right) \cdot \sqrt[2^i]{k} \cdot \left(\prod_{j=1}^{i} \sqrt[2^j]{\log n} \right).$$

Thus, for $i = \log(q) + 1$ iterations we obtain

$$\begin{aligned}
T^{\log(q)+1}(k) &= \left(\prod_{j=1}^{\log(q)+1} \sqrt[2^j]{n} \right) \cdot k^{\frac{1}{2q}} \cdot \left(\prod_{j=1}^{\log(q)+1} \sqrt[2^j]{\log n} \right) \\
&= n^{\sum_{j=1}^{\log(q)+1} 2^{-j}} \cdot k^{\frac{1}{2q}} \cdot (\log n)^{\sum_{j=1}^{\log(q)+1} 2^{-j}} \\
&= n^{1-\frac{1}{2q}} \cdot k^{\frac{1}{2q}} \cdot (\log n)^{1-\frac{1}{2q}} \\
&\overset{k \leq n^q}{\leq} n^{1-\frac{1}{2q}} \cdot \sqrt{n} \cdot (\log n)^{1-\frac{1}{2q}} \\
&\leq n^{3/2} \cdot \log n,
\end{aligned}$$

so at the beginning of the second phase $\mathcal{O}(n^{3/2} \cdot \log n)$ candidates are left w.h.p. □

4.4.2. Phase 2: Reducing Candidates to $\sqrt{n}$

In the next phase we are going to further reduce the size N of C to $\mathcal{O}(\sqrt{n})$. The idea is to let the second phase run in a constant amount of iterations, each of them reducing the size of the remaining candidates by factor $\Theta\left(\frac{\zeta \log n}{\sqrt{n}}\right)$, where $\zeta \in \Theta(\sqrt{\log n} \cdot \sqrt[4]{n})$. We will then show that only 5 iterations of the second phase are needed to reduce N to some value in $\mathcal{O}(\sqrt{n})$, w.h.p.

One iteration of the second phase is divided into 3 sub-phases $2a$, $2b$ and $2c$. In Phase 2a, we sample a set of $\Theta(\sqrt{n})$ candidates, which will then be sorted by their ranks in Phase 2b. We then use this sorting in Phase 2c where we choose two of these sampled candidates whose priorities form an interval that contains the k-th smallest element w.h.p. At the end of Phase 2c we remove all remaining candidates whose priorities are not contained in that interval.

Phase 2a: Choosing Representatives

We first sample a set $C' = \{e_1, \ldots, e_{n'}\} \subset C$ of n' candidates uniformly at random. To do this, the anchor propagates n and N to all nodes via the aggregation tree. Then each node u decides for each of its candidates $e \in u.C$ to be contained in the set C' with probability $\sqrt{n}/N$. We aggregate the number n' of chosen candidates to the anchor afterwards. Following this approach, we can show that $n' \in \Theta(\sqrt{n})$ w.h.p. due to Chernoff bounds.

Lemma 4.11. *After sampling has been done in Phase 2a, $n' \in \Theta(\sqrt{n})$ w.h.p.*

Proof. We show that n' does not significantly deviate from its expected value with the help of Chernoff bounds. For this, we show that n' is upper bounded by a constant factor of $\sqrt{n}$ (the proof that n' is lower bounded by a constant fraction of $\sqrt{n}$ works analogously). For each candidate $e \in C$ let X_e be the event that e has been sampled to be in C'. We have that $\Pr[X_e = 1] = \frac{\sqrt{n}}{N}$. For the sum $X = \sum_{e \in C} X_e$ of these events we have that $\mathbb{E}[X] = \sum_{e \in C} \frac{\sqrt{n}}{N} = \sqrt{n}$, so the size n' of the set C' is equal to $\sqrt{n}$ on expectation. Now choose $\delta = \frac{\sqrt{3c \log n}}{\sqrt[4]{n}}$ for a constant c. For n high enough it holds that $\delta \leq 1$. By Theorem 2.14(a) it follows that

$$\begin{aligned} \Pr[X \geq (1+\delta)\mathbb{E}[X]] &\leq \exp\left(\frac{-\left(\frac{\sqrt{3c \log n}}{\sqrt[4]{n}}\right)^2 \cdot \sqrt{n}}{3}\right) \\ &= \exp\left(-c \log n\right) \\ &< n^{-c} \end{aligned}$$

Therefore, w.h.p., n' is upper bounded by $2\sqrt{n}$. □

Phase 2b: Distributed Sorting

Our next goal is to compute the *order* of each candidate in C' when sorting them by their priorities (see Algorithm 7): For this we let the anchor assign a unique position $pos(e_i) \in \{1, \ldots, n'\}$ to each candidate $e_i \in C'$ via decomposition of the interval $[1, n']$ over the aggregation tree (similar to Phase 3 of Algorithm 5).

Every node routes each of its chosen $e_i \in C'$ to the node v_i responsible for position $pos(c_i)$ in the DHT (similar to Phase 4 of Algorithm 5). Then each node v_i generates n' copies of e_i and distributes them to n' other nodes in the following way: Let $b(v_i) = (v_{i,1}, \ldots, v_{i,d})$ be the first $d = \log n'$ bits of v_i's unique bit string according to the classical De Bruijn graph (recall that the aggregation tree is able to emulate routing in the classical de Bruijn graph due to Lemma 2.7). Node v_i stores a pair $([n'/2, n'/2], e_i)$ for itself and sends a pair $([1, n'/2 - 1], e_i)$ to the node with bit string $(0, v_{i,1}, \ldots, v_{i,d-1})$ and another pair $([n'/2 + 1, n'], e_i)$ to the node with bit string $(1, v_{i,1}, \ldots, v_{i,d-1})$. Repeating this process recursively until a node receives a pair $([a, b], e_i)$ with $a = b$ guarantees that n' nodes now hold a copy of e_i. Observe that this approach induces a (unique) tree $T(v_i)$ with root v_i and a height of at most $\log n' = \Theta(\log \sqrt{n}) = \Theta(\log n)$ when nodes remember the sender on receipt of a copy of e_i. Furthermore, there is no node serving as a bottleneck, which means that the number of trees that a node participates in is only constant on expectation:

Algorithm 7 Distributed Sorting

Input: $e_1, \ldots, e_{n'} \in C'$
Output: Order for each $e_1, \ldots, e_{n'}$ based on priorities

Algorithm (executed for each e_i)
1: Assign a unique position $pos(e_i) \in \{1, \ldots, n'\}$ to e_i
2: Route e_i to the node $v_i \in V$ responsible for $pos(e_i)$
3: Distribute n' copies $e_{i,1}, \ldots, e_{i,n'}$ of e_i over $v_{i,1}, \ldots, v_{i,n'} \in T(v_i)$
4: Route copy $e_{i,j}$ to $w_{i,j} \in V$ responsible for $h(i,j) \triangleright e_{i,j}$ and $e_{j,i}$ meet at $w_{i,j}$
5: **if** $\mathcal{P}(e_{i,j}) > \mathcal{P}(e_{j,i})$ **then**
6: Send $(1,0)$ to $v_{i,j}$, send $(0,1)$ to $v_{j,i}$
7: **else**
8: Send $(0,1)$ to $v_{i,j}$, send $(1,0)$ to $v_{j,i}$
9: Aggregate & combine vectors to the root node $v_i \in T(v_i)$
to obtain the order of e_i

Lemma 4.12. *Let $T(v_1), \ldots, T(v_{n'})$ be the unique trees as defined above and let $X_w = |\{T(v_i) \mid w \in T(v_i)\}|$. Then for all $w \in V$ it holds that $\mathbb{E}[X_w] = \Theta(1)$.*

Proof. Having N remaining candidates $e_1, \ldots, e_N$, there exist N unique trees $T(v_1)$, $\ldots, T(v_N)$ out of which we select n' uniformly at random, i.e.,

$$\Pr[\text{Tree } T(v_i) \text{ is selected}] = n'/N = \Theta(\sqrt{n}/N).$$

As each tree has height at most $\log n'$, the number of nodes in each tree is equal to

$$\sum_{i=0}^{\log n'} 2^i = \frac{1 - 2^{\log n' + 1}}{1 - 2} = 2 \cdot 2^{\log n'} - 1 = 2n' - 1.$$

Observe that since the root nodes of each tree are selected uniformly and independently at random and the tree height is only $\log n'$, all nodes in the trees are determined uniformly and independently at random. Thus, the probability that a node w is part of some tree T is equal to $\frac{2n'-1}{n} = \Theta(1/\sqrt{n})$. We can therefore compute the expected number of trees that w is part of: i.e., for $X_w = |\{T(v_i) \mid w \in T(v_i)\}|$ we get

$$\mathbb{E}[X_w] = N \cdot \Theta\left(\frac{\sqrt{n}}{N}\right) \cdot \Theta\left(\frac{1}{\sqrt{n}}\right) = \Theta(1).$$

□

Denote the element $e_{i,j}$ as the j-th copy of e_i, meaning that $e_{i,j}$ is the candidate e_i that is passed as part of the pair $([j,j], e_i)$ previously. Let $v_{i,j}$ be the node in $T(v_i)$ that received $e_{i,j}$. Then $v_{i,j}$ uses the pseudorandom hash function $h : \{1, \ldots, n'\}^2 \to [0,1)$ with $h(i,j) = h(j,i)$ for any $i, j \in \{1, \ldots, n'\}$ to route $e_{i,j}$ to the node $w_{i,j}$ in the DHT maintaining the key $h(i,j)$. Node $v_{i,j}$ also sends a reference to itself along with $e_{i,j}$. Once we have done this for all copies on all n' aggregation trees, a node $w_{i,j}$ has now received the following data: The copy $e_{i,j}$ along with the node $v_{i,j}$ and the copy

$e_{j,i}$ along with the node $v_{j,i}$. Thus, $w_{i,j}$ can compare the priorities $\mathcal{P}(e_{i,j})$ and $\mathcal{P}(e_{j,i})$ of $e_{i,j}$ and $e_{j,i}$. Based on the result of the comparison, $w_{i,j}$ sends a vector $(1,0)$ to $v_{i,j}$ and a vector $(0,1)$ to $v_{j,i}$ (in case $\mathcal{P}(e_{i,j}) > \mathcal{P}(e_{j,i})$) or a vector $(0,1)$ to $v_{i,j}$ and a vector $(1,0)$ to $v_{j,i}$ (in case $\mathcal{P}(e_{i,j}) < \mathcal{P}(e_{j,i})$). When $v_{i,j}$ receives a vector $(1,0)$ this means that there is one node in C' that has a smaller priority than e_i. Next, we aggregate and combine all these vectors to the root of each tree $T(v_i)$, using standard vector addition for combining. This results in v_i knowing the order of candidate e_i in C': If the combined vector at v_i is a vector $(L,R) \in \mathbb{N}^2$, then the order of e_i is equal to $L+1$.

Phase 2c: Reducing Candidates

In the next step, the anchor computes two candidates e_l and e_r such that we can guarantee w.h.p. that the element of rank k lies between the ranks of those candidates. For this, we consider the candidate $e_k \in C'$ for which the rank is closest to k on expectation, i.e., the candidate e_k with order $k \cdot \frac{n'}{N}$. Due to the way we computed the order of candidates in Phase 2b, there exists only one such element $e_k \in C'$. Now we move ζ candidates to the left/right of e_k in the ordering of candidates in C'. Let $e_l \in C'$ be the candidate whose order is equal to $l = \lfloor k \cdot \frac{n'}{N} - \zeta \rfloor$ and $e_r \in C'$ be the candidate whose order is equal to $r = \lceil k \cdot \frac{n'}{N} + \zeta \rceil$. In case $l < 1$ we just consider e_r and in case $r > n'$ we just consider e_l. For now, we just assume $l \geq 1$ and $r \leq n'$. We delegate e_l and e_r up to the anchor in the aggregation tree.

Once the anchor knows e_l and e_r, it sends them to all nodes in the aggregation tree. Now we compute the exact ranks of e_l and e_r in C via another aggregation phase. Each node u computes a vector $(l_u, r_u) \in \mathbb{N}^2$, where l_u represents the number of candidates in $u.C$ with a smaller priority than e_l and r_u represents the number of candidates in $u.C$ with a smaller priority than e_r. This results in the aggregation of a vector $(L,R) \in \mathbb{N}^2$ when using standard vector addition at each node in the aggregation tree. Once (L,R) has arrived at the anchor, it knows that $rank(e_l) = L+1$ and $rank(e_r) = R+1$. To finish the iteration, the anchor updates $v_0.k$ to $v_0.k - rank(e_l)$ and tells all nodes u in another aggregation phase to remove all candidates $e \in u.C$ with $rank(e) < rank(e_l)$ or $rank(e) > rank(e_r)$ and to aggregate the overall number k' of those candidates up to the anchor, such that it can update $v_0.N$. Then the anchor starts the next iteration (in case $v_0.N > \sqrt{v_0.n}$) or switches to the last phase of the protocol (in case $v_0.N < \sqrt{v_0.n}$).

We now show that this approach further reduces the number of candidates. First we want to compute the necessary number of shifts δ such that $rank(e_l) < k$ for e_l and $rank(e_r) > k$ for e_r holds w.h.p., as this impacts the number of candidates that are left for the next iteration of the second phase. For this we need the following technical lemma:

Lemma 4.13. *If $\zeta \in \Theta(\sqrt{\log n} \cdot \sqrt[4]{n})$, then w.h.p. $rank(e_l) < k$ and $rank(e_r) > k$.*

Proof. We just show $rank(e_l) < k$, as the proof for $rank(e_r) > k$ works analogously. Let $e_k \in C$ be the element with $rank(e_k) = k$. Let $X_i = 1$, if the candidate $e_i \in C$ with $rank(e_i) = i$ has been chosen to be in C' in Phase 2a. Let $X = \sum_{i=1}^{k} X_i$ be the number of elements with rank less than or equal to k that are chosen to be in C'.

Then $\mathbb{E}[X] = k \cdot \frac{\sqrt{n}}{N} \leq \sqrt{n}$. The probability that too few candidates with rank smaller than k have been chosen to be in C' should be negligible, i.e., $\Pr[X \leq \mathbb{E}[X] - \zeta] \leq n^{-c}$ for some constant c, where δ denotes the number of steps that we have to go to the left from the candidate with order $k \cdot \frac{\sqrt{n}}{N}$. In order to apply Chernoff bounds, we first compute $\varepsilon > 0$ such that $\Pr[X \leq (1-\varepsilon)\mathbb{E}[X]] = \Pr[X \leq \mathbb{E}[X] - \zeta]$. Solving the equation $(1-\varepsilon)\mathbb{E}[X] = \mathbb{E}[X] - \zeta$ for ε yields $\varepsilon = \zeta/\mathbb{E}[X]$. Using Chernoff bounds (Theorem 2.14(b)) on $\Pr[X \leq (1-\varepsilon)\mathbb{E}[X]]$ results in

$$\Pr[X \leq (1-\varepsilon)\mathbb{E}[X]] \leq \exp(-\varepsilon^2 \mathbb{E}[X]/2) \leq n^{-c}$$

for $2\varepsilon^2\mathbb{E}[X] = c \log n$, c constant. Solving this equation for ε leads to $\varepsilon = \sqrt{\frac{c \log n}{2\mathbb{E}[X]}}$. By solving the equation $\frac{\zeta}{\mathbb{E}[X]} = \sqrt{\frac{c \log n}{2\mathbb{E}[X]}}$ for ζ we get

$$\zeta = \sqrt{\frac{1}{2} \cdot c \log n \cdot \mathbb{E}[X]} \leq \sqrt{c \log n \cdot \sqrt{n}} = \Theta(\sqrt{\log n} \cdot \sqrt[4]{n}).$$

This means that choosing $\zeta \in \Theta(\sqrt{\log n} \cdot \sqrt[4]{n})$ suffices to guarantee that $rank(e_l) < rank(e_k)$ holds w.h..p. □

Using Lemma 4.13 we are now ready to show that only 5 iterations of Phase 2 suffice until the remaining number of candidates is within $\mathcal{O}(\sqrt{n})$.

Lemma 4.14. *After* 5 *iterations of the second phase,* $N \in \mathcal{O}(\sqrt{n})$ *w.h.p.*

Proof. Recall that by Lemma 4.10, we have $N = \mathcal{O}(n^{3/2} \cdot \log n)$ after the first phase. Consider the candidates e_l and e_r as determined by the anchor. Due to Lemma 4.13 it holds that $rank(e_l) < k < rank(e_r)$ and there are $\zeta \in \Theta(\sqrt{\log n} \cdot \sqrt[4]{n})$ candidates lying between e_l and e_r that are contained in C'; i.e., we consider the ordered sequence $e_l, e_{l+1}, \ldots, e_{r-1}, e_r$ of candidates in C'. We compute the number β of candidates that lie between two consecutive candidates $e_i, e_{i+1} \in C'$ such that the probability that all β candidates have not been chosen in Phase 2a becomes negligible, yielding an upper bound for the number of candidates lying between e_i and e_{i+1}. Recall that the probability that a candidate is chosen to be in C' is $\sqrt{n}/N$.

$$\begin{aligned}
\Pr[\beta \text{ candidates between } e_i \text{ and } e_{i+1} \text{ are not chosen}] &= (1 - \sqrt{n}/N)^{\beta} \\
&\leq \exp\left(-\frac{\sqrt{n}}{N} \cdot \beta\right) \\
&= n^{-c}
\end{aligned}$$

for $\beta = c \cdot \frac{N}{\sqrt{n}} \cdot \ln n = N \cdot \Theta\left(\frac{\log n}{\sqrt{n}}\right)$ and a constant c. Overall, it follows that N is reduced by factor $\Theta\left(\frac{\zeta \log n}{\sqrt{n}}\right)$ in each iteration of the second phase w.h.p. After five iterations of the second phase N is reduced to

$$\begin{aligned}
N \cdot \left(\frac{\zeta \log n}{\sqrt{n}}\right)^5 &\overset{Lemma\ 4.10}{=} n^{3/2} \cdot \log n \cdot \left(\frac{\zeta \log n}{\sqrt{n}}\right)^5 \\
&= \log^8(n) \cdot \sqrt{\log n} \cdot \sqrt[4]{n} \\
&= \mathcal{O}(\sqrt{n}).
\end{aligned}$$

□

Note that in case $l < 1$ (analogously $r > n'$), the set $\{e_1, \ldots, e_r\} \subset C'$ contains at most $\zeta \in \Theta(\sqrt{\log n} \cdot \sqrt[4]{n})$ candidates, so Lemma 4.14 still holds.

4.4.3. Phase 3: Exact Computation

The third and last phase computes the exact k-th smallest element out of the remaining candidates. This phase is basically just a single iteration of the second phase, with the exception that each remaining candidate is now chosen to be in C' in Phase 2a, leading to each candidate being compared with each remaining candidate. This immediately gives us the exact rank of each remaining candidate, as it is now equal to the determined order, so we are able to send the candidate that is the k-th smallest element to the anchor.

We are now ready to show Theorem 4.8:

Proof of Theorem 4.8. It is easy to see that in all three phases we perform a constant amount of aggregation phases for a constant amount of iterations. Note that in the second and third phase the time for a DHT-insert is $\mathcal{O}(\log n)$ w.h.p. due to Lemma 2.12. Also note that we perform the actions that have to be done in each of the generated $n' = \Theta(\sqrt{n})$ trees in parallel, resulting in a logarithmic number of rounds until the order of each chosen candidate is determined. As a single aggregation phase takes $\mathcal{O}(\log n)$ rounds w.h.p., we end up with an overall running time of $\mathcal{O}(\log n)$ w.h.p. for Algorithm 6.

For the congestion bound, note that the only time we generate more than a constant amount of congestion at nodes is in the second phase when routing the chosen candidates $e_i \in C'$ to the node v_i responsible for $pos(e_i)$ in $\mathcal{O}(\log n)$ rounds w.h.p. Thus, as each node chooses $\frac{\sqrt{n}}{N} \cdot \frac{N}{n} = \frac{\sqrt{n}}{n} = \mathcal{O}(1)$ of its candidates to be in C' on expectation, one can easily verify via Chernoff bounds and Lemma 2.8 that this generates a congestion of $\widetilde{\mathcal{O}}(1)$ w.h.p. With the same argumentation in mind, observe that the congestion generated for nodes that are part of at least one tree $T(v_i)$ is constant w.h.p., because each node participates in only two such trees on expectation (Lemma 4.12). Participation of node u in one of these trees means that u has to perform only one single comparison of priorities, leaving the congestion constant.

Finally, one can easily see that the message size is $\mathcal{O}(\log n)$ bits, because messages in our protocol contain only a constant amount of elements, where each element can be encoded by $\mathcal{O}(\log n)$ bits due to its priority being within $\{1, \ldots, n^q\}$. □

4.5. Arbitrary Priorities

We are now ready to demonstrate how to use the protocol for distributed k-selection from the previous section in order to implement a distributed priority queue for arbitrarily many priorities, i.e., for $\mathcal{P} = \{1, \ldots, n^q\}$, $q \in \mathbb{N}$ constant. In order to provide a scalable solution, we give up on the local consistency semantic (Definition 3.1(b)), which makes our protocol serializable instead of sequentially consistent.

4.5.1. Insert and DeleteMin

The general idea for processing requests is roughly the same as for the protocol with constant priorities. We first aggregate batches in the aggregation tree to the anchor, but instead of a batch representing both Insert and DeleteMin requests, we only aggregate the overall number of Insert requests or the overall number of DeleteMin requests. Consequently, we distinguish between separate *Insert Phases* and *DeleteMin Phases*. Algorithm 8 summarizes our protocol.

Algorithm 8 Distributed Priority Queue for an Arbitrary Amount of Priorities

Insert Phase

1: Aggregate the number $I \in \mathbb{N}$ of insertions to the anchor
2: $v_0.m \leftarrow v_0.m + I$
3: Broadcast the start of insertions over the tree
4: Store elements at random nodes

DeleteMin Phase

5: Aggregate the number $D \in \mathbb{N}$ of deletions to the anchor
6: Determine the element with rank D using Algorithm 6
7: Assign a unique key $k \in \{1, \ldots, D\}$ to the D most prioritized elements
8: Store these elements at the node maintaining the position $h(k)$
9: Assign a unique sub-interval $[a, b] \subset [1, k]$ to each node
that has to execute $b - a + 1$ DeleteMin requests
10: Fetch the elements stored at positions $\{a, \ldots, b\}$

Insert Phase. At the beginning of the Insert phase each node u generates a snapshot of the number of Insert operations stored in its local buffer and stores it in a variable $u.I$. Then the nodes aggregate all $u.I$'s to the anchor, using simple addition to combine two numbers $u.I$ and $u'.I$. When the anchor v_0 receives the aggregated value $v_0.I^+ = \sum_{u \in V} u.I$ at the end of the first phase, it updates $v_0.m$ and announces over the aggregation tree that nodes are now allowed to process Put requests on the DHT. For each element e that some node u wants to store in the DHT it assigns a key $k(e) \in \mathbb{N}$ generated uniformly at random and sends e to the node v that is responsible for $k(e)$ in the DHT. Once v has received e, it sends a confirmation message back to u. Upon receiving all confirmations for all its elements u switches to the DeleteMin phase.

DeleteMin Phase. Aggregation of DeleteMin requests works analogously to the Insert phase. At the end of the aggregation, the anchor v_0 receives a value D representing the number of DeleteMin requests to be processed. Now we use Algorithm 6 to find the element e with $rank(e) = D$. In order to assign a unique key $k \in \{1, \ldots, D\}$ to the D most prioritized elements, we proceed analogously as in Phase 3 of Algorithm 5 by decomposing the interval $[1, D]$ into sub-intervals. Each node u assigns such a key to all its stored elements that have a rank less than D. This can be determined by sending the priority of the k-th smallest element along with each sub-interval. The decomposition approach from Phase 3 of Algorithm 5 is also used to assign a unique sub-interval $[a, b] \subset [1, D]$ to each node that wants to execute $b - a + 1$ DeleteMin

requests. For the last step of our algorithm consider a node u that wants to issue d DeleteMin requests on the priority queue and consequently got assigned to the sub-interval $[a, b]$ such that $d = b - a + 1$. Then u generates a $\mathsf{Get}(h(k), u)$ request for each key $k \in \{a, \ldots, b\}$ to fetch the element that previously got stored in the DHT at that specific position. This way, each DeleteMin request got a value returned so nodes can then proceed with the Insert phase afterwards.

4.5.2. Analysis

The following analysis of the distributed priority protocol for arbitrarily many priorities follows the same structure as the previous ones.

Lemma 4.15. *The distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(poly(n))$ satisfies serializability.*

Proof. To show serializability, we define the total order $\prec$ for all priority queue requests, whose serial execution is equivalent to the distributed execution of requests in Algorithm 8. Recall that S is the set of all requests to be issued on the priority queue. In order to define $\prec$ we assign a unique virtual value to each request op via the function $\phi : S \to \mathbb{N}$. We divide S into pairs (S_I^i, S_D^i), where S_I^i contains all Insert requests processed in the i-th Insert phase and S_D^i contains all DeleteMin requests processed in the i-th DeleteMin phase. Obviously, $\bigcup_{i=1}^{T} S_I^i \cup S_D^i = S$ where T denotes the overall number of Insert and DeleteMin phases during the lifetime of the priority queue.

For a subset S_I^i containing k Insert requests we fix a randomly chosen permutation of the requests (i.e., $S_I^i = (\mathsf{ins}_1, \ldots, \mathsf{ins}_k)$) and set $\phi(\mathsf{ins}_j) = i \cdot j$ for all $1 \leq j \leq k$. For the corresponding set $S_D^i = \{\mathsf{del}_1, \ldots, \mathsf{del}_l\}$ containing l DeleteMin requests note that each request gets assigned a unique key out of $\{1, \ldots, l\}$ by the anchor in case the priority queue contains at least l elements. Assume that the request del_j got assigned the key j by the anchor. Then we set $\phi(\mathsf{del}_j) = j + i \cdot k$. In case the priority queue contains less than l elements, say only l' elements, we order the requests in S_D^i such that the first l' requests got assigned a key by the anchor, while the other requests did not get assigned a key (and thus return $\perp$). Afterwards, we set $\phi(\mathsf{del}_j) = j + i \cdot k$ for all $j \leq l$ accordingly.

The total order $\prec$ is then defined by the total order of all requests induced by ϕ, i.e., $\mathsf{op}_i \prec \mathsf{op}_j$ if and only if $\phi(\mathsf{op}_i < \phi(\mathsf{op}_j)$. This is in accordance with our protocol because we handle Insert and DeleteMin requests in separate phases, which means that we wait until all Insert requests have been processed before we start processing all DeleteMin requests. Hence, our protocol satisfies serializability. □

Lemma 4.16. *The distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(poly(n))$ satisfies priority queue consistency.*

Proof. We prove each property of Definition 4.1 separately using the total order $\prec$ as defined in the proof of Lemma 4.15.

(a) Let $(\mathsf{ins}(u), \mathsf{del}(v)) \in M$. Then the Insert phase where $\mathsf{ins}(u)$ got inserted in the priority queue occurred before the DeleteMin phase where $\mathsf{del}(v)$) has been processed, so it follows $\mathsf{ins}(u) \prec \mathsf{del}(v)$.

(b) Let $\mathsf{del}(u), \mathsf{ins}(v) \notin M$ and assume to the contrary that $\mathsf{ins}(v) \prec \mathsf{del}(u)$. Again, by definition of $\prec$, the Insert phase where $\mathsf{ins}(v)$ got inserted in the priority queue occurred before the DeleteMin phase where $\mathsf{del}(u)$) has been processed. Since $\mathsf{ins}(v)$ is not matched, the element e inserted via this request is still contained in the priority queue at the time where $\mathsf{del}(u)$) is processed. Therefore $\mathsf{del}(u)$) has to get matched to $\mathsf{ins}(v)$, which is a contradiction.

(c) Let $(\mathsf{ins}(u), \mathsf{del}(v)) \in M$ and assume that there exists $\mathsf{ins}(w) \notin M$ such that $\mathsf{ins}(w) \prec \mathsf{del}(v)$ and $\mathcal{P}(\mathsf{ins}(w)) < \mathcal{P}(\mathsf{ins}(u))$. Since $\mathsf{ins}(w) \prec \mathsf{del}(v)$ holds, the elements inserted via $\mathsf{ins}(u)$ and $\mathsf{ins}(w)$ are still contained in the priority queue once $\mathsf{del}(v)$ is processed in the subsequent DeleteMin phase. Thus, the element inserted via $\mathsf{ins}(w)$ should have been taken out of the priority queue first since $\mathcal{P}(\mathsf{ins}(w)) < \mathcal{P}(\mathsf{ins}(u))$ holds. This implies that $\mathsf{ins}(w)$ gets matched (either to $\mathsf{del}(v)$ or some other DeleteMin request), which is a contradiction to our assumption that $\mathsf{ins}(w) \notin M$.

This concludes the proof of the lemma. □

The next lemma serves as a proof for the number of rounds needed to process priority queue requests successfully:

Lemma 4.17. *The Insert phase and the DeleteMin phase finish after $\mathcal{O}(\log n)$ rounds w.h.p.*

Proof. The runtime of the Insert phase follows from Corollary 2.10 and Lemma 2.12. We argue that each step in the DeleteMin phase listed in Algorithm 8 takes at most $\mathcal{O}(\log n)$ rounds w.h.p. Aggregating the number D of deletions to the anchor can be done in a single aggregation phase. Determining the element with rank D takes $\mathcal{O}(\log n)$ rounds w.h.p. due to Theorem 4.8. Assigning keys to the elements that have to be removed is done via a broadcast down the aggregation tree, so it takes $\mathcal{O}(\log n)$ rounds w.h.p. (Corollary 2.10). Delivering these elements to the node responsible for the corresponding key takes $\mathcal{O}(\log n)$ rounds w.h.p. (Lemma 2.7). Decomposing the interval $[a, b]$ into sub-intervals and assigning these sub-intervals to all nodes again is done via a broadcast down the aggregation tree, so it takes $\mathcal{O}(\log n)$ rounds w.h.p. Finally, fetching the elements stored at the corresponding positions in the DHT is equivalent to issuing Get requests on the DHT, so it takes $\mathcal{O}(\log n)$ rounds w.h.p. (Lemma 2.12). □

Lemma 4.18. *The distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(poly(n))$ has congestion $\widetilde{\mathcal{O}}(\Lambda)$.*

Proof. At the beginning an Insert phase, each node u has at most $\lambda(u) \cdot \mathcal{O}(\log n) = \widetilde{\mathcal{O}}(\lambda(u))$ Insert requests buffered, since the previous phase lasted for $\mathcal{O}(\log n)$ rounds (Lemma 4.17) and u could have generated at most $\lambda(u)$ requests per round. For each of those requests, u delegates the element to a randomly chosen node, resulting in u having to process $\widetilde{\mathcal{O}}(\lambda(u))$ requests at once. Since each of these delegations needs $\mathcal{O}(\log n)$ rounds w.h.p. (Corollary 2.10) to finish and the aggregation tree only generates congestion up to a polylogarithmic factor (Lemma 2.8), the lemma follows for the Insert phase.

For the DeleteMin phase, the steps of Algorithm 8 where we use the aggregation tree have no impact on the upper bound for the congestion. For selecting the element with rank D, the congestion is $\widetilde{\mathcal{O}}(1)$ due to Theorem 4.8. In the last step at which nodes fetch data from the DHT we can use the same argumentation as for the Insert phase, so the lemma follows. □

Lemma 4.19. *Messages generated in the distributed priority queue with $|\mathcal{P}| \in \mathcal{O}(poly(n))$ consist of $\mathcal{O}(\log n)$ bits.*

Proof. First note that the number of priority queue requests can only be polynomial in n. Due to Lemma 4.17 each node may generate up to $\Lambda \in \mathcal{O}(poly(n))$ new requests for $\log n$ rounds until the next phase is started. Thus, aggregating the number of priority queue requests to the anchor yields a message size of $\mathcal{O}(\log n)$. Note that in order to decompose and assign (sub-)intervals to all nodes we only need to store a single interval in each message (similar to Phase 3 of Algorithm 5). Keeping this argumentation in mind and the fact that our protocol for distributed k-selection uses only $\mathcal{O}(\log n)$-bit messages (Theorem 4.8), one can easily see that Algorithm 8 uses only $\mathcal{O}(\log n)$-bit messages. □

The following theorem summarizes the results obtained in this section.

Theorem 4.20. *Algorithm 8 implements a distributed priority queue for an arbitrary amount of priorities with the following properties:*

(a) *The distributed priority queue satisfies serializability and priority queue consistency.*

(b) *Each priority queue request is finished after at most $\mathcal{O}(\log n)$ rounds w.h.p.*

(c) *The distributed priority queue has congestion $\widetilde{\mathcal{O}}(\Lambda)$.*

(d) *Messages generated in the distributed priority queue consist of $\mathcal{O}(\log n)$ bits.*

(e) *If the distributed priority queue contains m elements, each node stores m/n elements on expectation.*

CHAPTER 5

Conclusion and Outlook of Part I

To conclude the first part of this thesis, we sum up our results on distributed data structures and give some more details regarding further properties and potential future work in this area.

Conclusion

Table 5.1 summarizes the most important properties of all the distributed data structures presented in the first part of this thesis. Note that all of these data structures also support *fairness* due to the underlying distributed hash table, so all nodes store the same amount of elements on expectation. Furthermore, all of our protocols support Join requests in $\mathcal{O}(\log n)$ rounds and Leave requests in $\mathcal{O}(1)$ rounds due to Theorems 3.18 and 3.19.

Data Structure	Semantics	Requests	Congestion	Message Size
Queue	SC+QC	$\mathcal{O}(\log n)$	$\widetilde{\mathcal{O}}(\Lambda)$	$\mathcal{O}(\Lambda \log^2 n)$
Stack	SC+StC	$\mathcal{O}(\log n)$	$\widetilde{\mathcal{O}}(\Lambda)$	$\mathcal{O}(\log n)$
P-Queue, $\lvert\mathcal{P}\rvert \in \mathcal{O}(1)$	SC+PQC	$\mathcal{O}(\log n)$	$\widetilde{\mathcal{O}}(\Lambda)$	$\mathcal{O}(\Lambda \log^2 n)$
P-Queue, $\lvert\mathcal{P}\rvert \in \mathcal{O}(poly(n))$	S+PQC	$\mathcal{O}(\log n)$	$\widetilde{\mathcal{O}}(\Lambda)$	$\mathcal{O}(\log n)$

Table 5.1.: Table for all distributed data structures and their properties (semantical guarantees, runtime for requests, congestion and maximum message size in bits) obtained via Theorems 3.11, 3.17, 4.6 and 4.20. In the semantics column, *SC* stands for *sequential consistency*, *QC* for *queue consistency*, *StC* for *stack consistency*, *PQC* for *priority queue consistency* and *S* for *serializability*. All bounds on the runtime hold with high probability.

In scenarios where the rate Λ in which requests on the data structure are generated is very high, the distributed stack and the priority queue for arbitrarily many priorities perform very well, since the size of messages does not depend on Λ but only on n. The reason why the size of messages in the distributed queue protocol is higher than just $\mathcal{O}(\log n)$ is that the queue satisfies local consistency in addition to serializability, so when aggregating batches to the anchor, we have to respect the local order in which requests are generated by the nodes. One way to reduce the size of messages to $\mathcal{O}(\log n)$ in the distributed queue would therefore be to give up on local consistency and only require serializability. Our protocol can easily be modified to work this way, as one only has to set up separate Enqueue and Dequeue phases in the same manner as we did for the distributed priority queue with arbitrarily many priorities. The same strategy could be applied to the priority queue with a constant amount of

priorities, by which we obtain the same properties as for the priority queue for an arbitrary amount of priorities. While we believe that both, Algorithms 5 and 8 are viable in practice with the above modifications, we recommend running Algorithm 5 when there is only a constant amount of priorities as the protocol for distributed k-selection induces a higher constant factor behind the $\mathcal{O}(\log n)$ runtime bound. However, as soon as we need a priority queue with more than a constant amount of priorities, we recommend running Algorithm 8.

Aside from the above argumentation, improving any of the protocols from Table 5.1 probably requires different techniques than we used in this thesis, potentially requiring a different network topology of the processes. This then comes with the problem of having to adjust the protocols for Join and Leave as they rely on the LDB network structure and abuse the fact that the (virtual) nodes only form a simple ring, which makes updating the topology fairly easy compared to other graphs where the degree of nodes is larger than 2.

Outlook

Heterogeneous Distributed Data Structures. So far, we assumed *homogeneity*, i.e., we assumed that all processes have the same capabilities, for example, regarding their storage and their bandwidth. An interesting direction for future research could be to assume *heterogeneity*. Here, one could assume that processes now have different capabilities in terms of storage space and/or bandwidth.

Processes with a higher storage capability are able to store more elements of the data structure. As a consequence, one could aim to let a process u, whose storage capability is $p\%$ of the total storage, receive $p\%$ of the elements of the data structure (this would replace the fairness definition used in this thesis). As a starting point, one could have a look at already existing versions for heterogeneous distributed hash tables like [KKS13].

The bandwidth of a processes is an indicator on how many messages the process can send and receive within a specific amount of time. One could try to have processes with a higher bandwidth take more responsibility in the distributed protocol, for example by choosing a different network topology. Combined with a suitable protocol, this could result in a potential speedup in the worst-case runtime for data structure requests.

Further Distributed Data Structures. While queues, stacks and priority queues are fundamental data structures, there are many other data structures that could require effective distributed implementations. The most relevant example right now is probably the distributed ledger that is often used in blockchain protocols. Blockchain protocols are able to provide a total order on transactions issued by all the nodes, mostly via the use of expensive distributed consensus protocols. We believe that using the distributed queue could potentially be helpful to establish such a global transaction in a more cost-efficient manner.

Further data structures for which there exist concurrent algorithms already could also ask for distributed implementations. Consider, for example, search trees [AS16] or pools [AHS94].

Fault-Tolerant Distributed Data Structures. Another potentially interesting direction for future research is to investigate if one can make our distributed protocols fault tolerant. Here one could investigate if there are possibilities to cope with byzantine nodes that could maliciously work against other honest nodes, for example by flooding the system with a large amount of corrupted requests or by sending wrong batches to disrupt the protocols. A possible approach to tackle the latter problem could be to form a committee of nodes that try to reach consensus on the correct batch that arrives at the anchor. One has to make sure that the number of honest nodes outnumber the byzantine nodes in the committee the data structure. Similar strategies (called *sharding*) are also used to perform tasks such as improving the performance and scalability of distributed ledgers, see for example *Elastico* [Luu+16] or *OmniLedger* [Kok+18].

Making our data structures self-stabilizing could also be another possibility in order to automatically repair the system without having to rely on external intervention. In the second part of this thesis we present self-stabilizing protocols that are able to automatically repair the network topology that is formed by the processes. Similar techniques have to be applied to our distributed data structures (in fact, the linearized De Bruijn network has already been made self-stabilizing [RSS11]) – not only to repair the topology (i.e., the LDB network) but also to make sure that elements of the data structure are stored by the correct node and that one can detect whether insertion and deletion requests are processed correctly on the correct data set all the time. A similar problem has already been investigated in [KS18] where the authors presented a self-stabilizing protocol for the embedding of a Patricia trie into a distributed system. This makes us confident that one is able to construct self-stabilizing protocols for our distributed data structures in a similar manner.

Part II.

Self-Stabilizing Overlay Networks

CHAPTER 6

Preliminaries

In the second part of this thesis we study self-stabilizing overlay networks. An overlay network is a virtual network that is built on top of the physical network. A single edge in an overlay network may correspond to a path in the underlying physical network. Overlay networks are used in distributed systems, especially in peer-to-peer networks of client-server applications. As the underlying physical network may not be coordinated (i.e., the edges do not resemble a certain class of graph topologies), overlay networks improve the coordination among the nodes. Depending on the application, different topologies have to be considered. For example, to provide short routing paths, a network with a low diameter would be preferable, such as a clique. In case the location of the nodes plays a role, such as in wireless networks, we could use Delaunay graphs, unit disk graphs or quadtrees. Finally, overlay networks can also be used to support distributed applications such as publish-subscribe systems.

In this part of the thesis, we are specifically concerned with making the overlay networks *self-stabilizing*, meaning that we require the network to repair itself once it does not form the desired topology anymore due to the results of faults or adversarial attacks. These kinds of errors are common in distributed systems. The repair process should be done only by the participants of the network and without external intervention. More specifically, when the system starts in any arbitrary initial state, we want the system to arrive in a legitimate state within a finite amount of time. Also, once the system has reached a legitimate state, it should stay in legitimate states thereafter in case the set of nodes remains static and no faults (for example, blackout of nodes or message loss) occur.

In this thesis we present novel distributed algorithms for repairing overlay networks and build them into specific topologies that serve one of the above mentioned applications. Specifically, we show how one can manipulate the edges of the overlay network in order to build *generalized De Bruijn graphs*, *quadtrees* and *supervised publish-subscribe systems* in a self-stabilizing manner. Such algorithms fall in the category of *topological self-stabilization*. Our protocols rely on some well-known primitives for the manipulation of overlay edges, as well as on self-stabilizing protocols for sorted lists and rings, which we are going to introduce in this chapter.

Outline of This Chapter. We first state the model in Section 6.1. Then we give a precise definition for self-stabilization and also introduce the four universal primitives (Introduce, Forward, Merge and Fusion) for manipulating overlay networks (Section 6.2). Afterwards, we give an overview of related work on self-stabilization and, more specifically, topological self-stabilization (Section 6.3). We finish this chapter by presenting the well-known protocol for self-stabilizing sorted lists (Section 6.4) and its extension to self-stabilizing sorted rings (Section 6.5). These protocols serve as a base for the protocols in the next chapters.

6.1. Model

We first define the overlay network similar to Definition 2.1 from the first part of this thesis.

Definition 6.1. *The* overlay network *is a directed graph $G = (V, E)$ for a fixed set of n nodes with the following properties:*

(a) *Each node $u \in V$ represents a single process and is identified by a unique* identifier*, or short* ID*, denoted by $id(u) \in \mathbb{N}$.*

(b) *Each node $u \in V$ maintains local protocol-based variables and has a* channel *$u.Ch$, which is a system-based variable that contains incoming messages. A channel may store any finite number of messages. Messages are never duplicated nor do they get lost in the channel.*

(c) *If a node u has the reference of some other node v stored in one of its local variables, u can send a message m to v by putting m into $v.Ch$. There is a directed edge $(u, v) \in E$ whenever u stores a reference of v in one of its local variables or there exists a message in $u.Ch$ carrying the reference of v. In the former case, we call that edge* explicit *(drawn solid in figures) and in the latter case we call that edge* implicit *(drawn dashed).*

Nodes may execute actions, which we define in the following:

Definition 6.2. *A node may execute the following types of* actions*:*

(a) *The first type is of the form*

$$\langle label \rangle(\langle parameters \rangle) : \langle command \rangle,$$

where label is the name of that action, parameters defines the set of parameters and command defines the statements that are executed when calling that action.

(b) *The second type for an action is of the form*

$$\langle label \rangle : \langle guard \rangle \rightarrow \langle command \rangle,$$

where label and command are defined the same as above and guard is a predicate over local variables. Such an action may be executed only if its guard is true.

Actions of the first type may be initiated locally or remotely; i.e., every message that is sent to a node has the form $\langle label \rangle(\langle parameters \rangle)$[1]. When a node u executes the action represented by the message m, it removes m from $u.Ch$. An action whose guard is simply *true* is called Timeout and is executed periodically by each node.

Definition 6.3. *The* system state *is an assignment of values to every node's variables and messages to each channel.*

[1] Having a node u send a message of the form $\langle label \rangle(\langle parameters \rangle)$ to some node v is denoted by $v \leftarrow \langle label \rangle(\langle parameters \rangle)$ in the pseudocode for the node u.

If an action can be executed by some node u (meaning that either there is a message in $u.Ch$ that requests to call that action, or its guard is *true*), we call that action *enabled* for u.

Definition 6.4. *A* computation *is an infinite sequence* $(s_0, s_1, \ldots)$ *of system states, where the state* s_{i+1} *is reached from its previous state* s_i *by executing an action that is enabled in* s_i*. A state* s' *is* reachable *from some state* s *if we end up in state* s' *via a sequence of action executions when starting in state* s*. We call the first state* s_0 *of a given computation the* initial state*.*

We assume *fair message receipt*, meaning that every message that is contained in some channel is eventually processed and thus its corresponding action is executed eventually. Furthermore, we assume *weakly fair action execution*, so if an action is enabled in all but finitely many states of a computation, then this action is executed infinitely often. Consider the Timeout action as an example for this; i.e., Timeout is executed periodically by each node. We place no bounds on message propagation delay or relative node execution speed, which means that we allow fully asynchronous computations and non-FIFO message delivery.

Our protocols do not manipulate node identifiers and thus operate on them only in *compare-store-send* mode: i.e., the nodes are only allowed to compare node identifiers, store them in their local memory or send them in a message. We assume that there are no corrupted node identifiers (i.e., IDs of unavailable nodes) in the initial state of the system. Identifying corrupted IDs requires failure detectors, which is not within the scope of this thesis. Since our self-stabilizing protocols just deal with IDs in a compare-store-send manner, this implies that node IDs have to be non-corrupted for all computations, as has been shown in [NNS13, Theorem 2]. Nevertheless, the node channels may initially contain an arbitrary finite number of messages containing false information. We call these messages *corrupted*, and we will argue that eventually the system reaches a state without corrupted messages when starting at some initial state.

6.2. Self-Stabilization and Primitives for Overlay Networks

We are now ready to precisely define the notion of *self-stabilization*.

Definition 6.5 (Self-Stabilization)**.** *A protocol is* self-stabilizing *w.r.t. a set of legitimate states if it satisfies the following two properties:*

(a) Convergence*: Starting from an arbitrary system state, the protocol is guaranteed to reach a legitimate state.*

(b) Closure*: Starting from a legitimate state, the protocol remains in legitimate states thereafter.*

In order for our distributed algorithms to work, we require the directed graph G containing all explicit and implicit edges to stay at least weakly connected at every point in time. A directed graph $G = (V, E)$ is *weakly connected* if the undirected version of G is connected; i.e., for two nodes $u, v \in V$ there is a path from u to

v in the undirected version of G. Once G consists of multiple weakly connected components, they cannot be connected to each other anymore as it has been shown in [NNS13, Theorem 1] for compare-store-send protocols. The self-stabilizing protocol will work on each connected component individually instead.

Defining the set of legitimate states is crucial in order to prove for a protocol that it is self-stabilizing. As we are dealing with *topological self-stabilization* in this thesis for most of the time, the legitimate state is usually defined by the graph induced by the explicit edges, which means that our goal is to transform any initially weakly connected graph to a graph where the explicit edges form some desired topology. We give precise definitions for the set of legitimate states for all of our protocols in the corresponding chapters.

In order to manipulate the edges of the network, we rely on four simple primitives, which we present in the following.

Definition 6.6. *The following primitives are allowed to manipulate the edges of the graph $G = (V, E)$:*

(a) *Introduce: If a node u has references to nodes v and w, then u may* introduce *w to v by sending a message containing the reference of w to v. For the special case that $u = w$, we say that u* introduces itself *to v.*

(b) *Forward: If a node u has references to nodes v and w with $v \neq w$, then u may* forward *w to v by sending a message containing the reference of w to v. Afterwards, u removes the reference to w from its local storage. We may also say that u forwards its edge to w to the node v.*

(c) *Merge: If a node u has references to nodes v and w such that $v = w$, then u may* merge *these references by keeping only one in its local storage and deleting the other.*

(d) *Invert: If a node u has a reference to some node v, then u may* reverse *the connection to v by sending a message containing its own reference to v and deleting the reference to v from its local storage afterwards.*

Consider Figure 6.1 for an illustration of the primitives. Intuitively, the roles of these primitives is that Introduce is used to increase the number of edges, Forward is used to separate two nodes from each other, Merge is used to reduce the number of edges and Invert is used to make a node unreachable.

It is easy to see that these four primitives preserve the weak connectivity of the graph at any point in time. Also, as shown in [KSS17], these primitives together are known to be *universal*:

Theorem 6.7 (Universality, [KSS17])**.** *The primitives Introduce, Forward, Merge and Invert are universal: They can turn any weakly connected graph $G = (V, E)$ into any other weakly connected graph $G' = (V, E')$.*

Note that in order to construct any *strongly* connected graph, the primitives Introduce, Forward and Merge already suffice. Finally, it has also been shown in [KSS17] that the four primitives are needed in order to provide universality.

Theorem 6.8 (Necessity, [KSS17])**.** *Introduce, Forward, Merge and Invert are necessary for universality.*

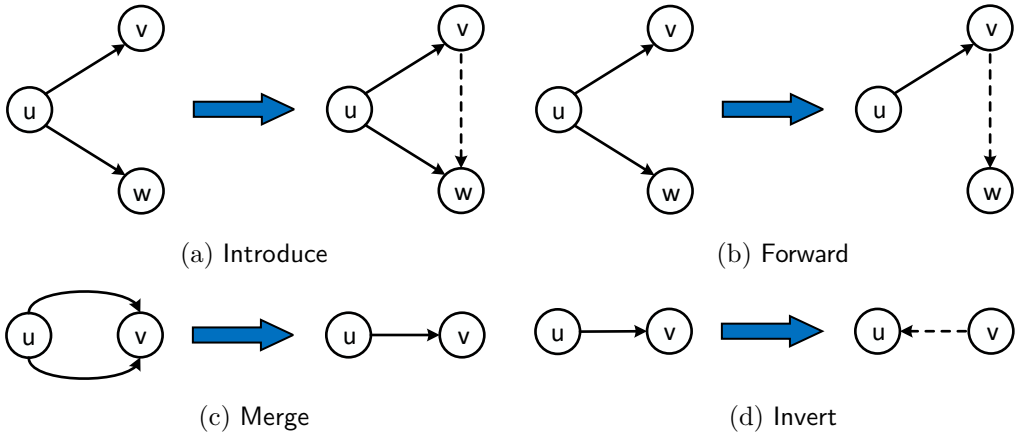


Figure 6.1.: Illustration of the four primitives. Straight lines indicate explicit edges, dashed lines indicate implicit edges.

6.3. Related Work

We cover related work from the area of self-stabilization and, more specifically, from the area of topological self-stabilization. Further related work that is more relevant to the protocols in the following chapters will be covered in separate sections.

Self-Stabilization. The concept of self-stabilizing algorithms for distributed systems goes back to the year 1974, when E. W. Dijkstra introduced the idea of self-stabilization in a token-based ring [Dij74]. This seminal work marked the beginning of a whole new research area. Researchers started to create self-stabilizing protocols for various settings, including (but not limited to) classical problems in distributed computing such as maximal matching [HH92; Man+09; Coh+16], coloring [GK93; SS93; LC10; LL14] or clock synchronization [DW04; BDH08; KL19; FKS20]. Therefore it is not surprising that Dijkstra's seminal work on self-stabilization has been referred to as "his most brilliant work" by Leslie Lamport in his ACM PODC keynote address [Lam85]. For a survey on self-stabilizing algorithms consider the book by Dolev [Dol00].

Topological Self-Stabilization. The first topologies that were investigated have been self-stabilizing sorted lists [ORS07; Gal+14] and rings [SR05; CF05]. One fundamental technique that lays the foundation for many other works in this area (including the ones presented in this thesis) is *linearization*. The idea is that a node always keeps the nodes that are closest to it (w.r.t. to some total order of the nodes) from its local point of view and forwards all other edges away, such that in a legitimate state each node only knows its closest left and right neighbor. Using this technique, one is able to sort the nodes in a self-stabilizing manner and thus arrange all nodes in a sorted list (see the next section for more details).

Self-stabilizing protocols that rely on sorted lists or rings are, for example, chord graphs [KKS14; Ben+13] and small-world networks [KKS12].

Much work has also been done on trees [AK93; Hér+06; Clé+08; DK08; AW07], but the scenario here is that the network graph is static and a self-stabilizing solution

computes a set out of these static edges that together form a (spanning-) tree. A topologically self-stabilizing solution is presented by Götte et al. [GSS18]. They show how to compute a minimum spanning tree when given an overlay network as an initially weakly connected graph $G = (V, E)$ and a tree metric $d_T : V^2 \to \mathbb{R}^+$ that assigns a weight to each edge that can possibly exist in the overlay network. The idea here is similar to the linearization technique, as each node forwards an edge that can be replaced by a more light-weight edge. This makes the protocol *locally checkable*, meaning that each node is able to check if its own state is illegal from its local point of view. Note that nodes cannot locally decide if the entire system is in a legitimate state. However, local checkability comes with the advantage that we do not need to invest global communication to check if the system is in a legitimate state – the system is in a legitimate state once all nodes are in a legitimate state from their local point of view.

There exist self-stabilizing protocols for skip lists and skip graphs [NNS13; CNS12]. Unfortunately, skip graphs cannot be checked locally for correctness, as nodes are not able to deduce whether the skip graph is in a correct state based only on their local neighborhood. Nevertheless, Jacob et al. [Jac+14] came up with a self-stabilizing protocol for a SKIP$^+$ graph that extends the skip graph by additional edges that allow for local checkability. The protocol also relies on a sorted list that is built at each of the $\mathcal{O}(\log n)$ levels of the skip graph.

A self-stabilizing clique has been proposed in [KKS15]. The idea of the protocol is to first collect the references of all nodes at the node with maximum identifier and then broadcast this information via a sorted list.

Having access to a clique can also be useful when constructing self-stabilizing protocols for any desired topology, as has been demonstrated by Berns et al. [BGP13] in their *Transitive Closure Framework*. Here the idea is to construct a clique once at least one node found out that it is not in a legitimate state. Once the clique has been constructed, each node is able to compute its correct set of neighbors locally and thus remove all other edges via the Forward primitive. Since such a protocol lets the degree of nodes become quite large temporarily, the Avatar-framework has been proposed [Ber15], which is also able to create different families of graphs while bounding the amount by which the degree of nodes increases to a polylogarithmic factor on expectation only.

Another important aspect in topological self-stabilization includes investigating the *leave problematic*, where one asks for self-stabilizing protocols that are able to safely exclude nodes that want to leave the system. It is important to note that nodes cannot simply leave the system on their own, as such an action may disconnect the graph. Therefore, one may ask if there exist self-stabilizing protocols that are able to exclude leaving nodes from the system. Here it has been shown that in general it is impossible for local control protocols to solve this problem reliably [For+14], so one has to make use of oracles. However, if nodes that want to leave are just allowed to be put into a *sleeping* state while still remaining in the system, one can construct a self-stabilizing protocol that guaranteed that all leaving nodes are in the sleeping state once the system reaches a legitimate state [KSS17].

For more insights on the techniques used in topological self-stabilization, we recommend our survey on algorithms for self-stabilizing overlay networks [FSS20].

6.4. Self-Stabilizing Sorted Lists

In this section we describe the well-known protocol for the self-stabilizing sorted list, called BuildList.

Given an initially weakly connected directed graph $G = (V, E)$ and a total order $\prec$ on the nodes, the goal of BuildList is to transform G into a line graph $G' = (V, E')$ such that the explicit edges in E' form a sorted list.

In this section we define $u \prec v$ if and only if $id(u) < id(v)$. We want to emphasize that $\prec$ may be defined differently in the following chapters, but this does not impact the BuildList protocol as it works on any predefined total order. The following definition introduces some additional terminology:

Definition 6.9. *A node $u \in V$ is* left *to $v \in V$, if $u \prec v$, otherwise u is* right *to v. The* closest left neighbor *of some node $u \in V$ is the node v such that*

$$v \prec u \wedge \forall w \neq v, w \prec u : w \prec v.$$

Similarly, the closest right neighbor *of some node $u \in V$ is the node v such that*

$$u \prec v \wedge \forall w \neq v, u \prec w : v \prec w.$$

Before we describe BuildList, we state the variables that have to be maintained by each node $u \in V$:

Definition 6.10. *For the BuildList protocol, each node $u \in V$ maintains the following variables:*

(a) *$u.left \in V \cup \{\perp\}$: Stores u's current left neighbor in the sorted list.*

(b) *$u.right \in V \cup \{\perp\}$: Stores u's current right neighbor in the sorted list.*

On the basis of the above variables, we define the legitimate state that has to be reached by BuildList:

Definition 6.11 (Legitimate State for BuildList)**.** *The system is in a* legitimate state *for BuildList if the following properties hold for each node $u \in V$:*

(a) $u.left$ stores the closest left neighbor of u or $\perp$ if there exists no node that is left to u.

(b) $u.right$ stores the closest right neighbor of u or $\perp$ if there exists no node that is right to u.

Figure 6.2 illustrates a possible legitimate state for the sorted list.

The idea of BuildList is that each node u keeps its closest left and right neighbors on the basis of its local information. All other node references to nodes v should be forwarded by u to either $u.left$ or $u.right$, depending on whether $u \prec v$ or $v \prec u$ holds. Since u knows the reference of v, it also knows $id(v)$, so it is able to compare $id(v)$ to its own ID.

BuildList (Algorithm 9) consists of the actions Timeout and Linearize(v). Timeout is executed periodically at each node and Linearize(v) can be called locally or remotely.

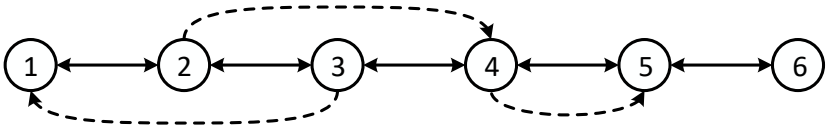


Figure 6.2.: Possible legitimate state for BuildList. Note that a legitimate state only requires the explicit edges (straight edges) to form a sorted list.

Algorithm 9 The BuildList Protocol (executed by each node $u \in V$)

```
1:  Timeout: true →
2:      if u ≺ u.left then
3:          Linearize(u.left)
4:          u.left ←⊥
5:      if u.right ≺ u then
6:          Linearize(u.right)
7:          u.right ←⊥
8:      u.left ← Linearize(u)
9:      u.right ← Linearize(u)

10: Linearize(v):
11:     if v ≺ u.left then
12:         u.left ← Linearize(v)
13:     if u.left ≺ v ≺ u then
14:         v ← Linearize(u.left)
15:         u.left ← v
16:     if u ≺ v ≺ u.right then
17:         v ← Linearize(u.right)
18:         u.right ← v
19:     if u.right ≺ v then
20:         u.right ← Linearize(v)
```

In Timeout a node u first performs a consistency check for its variables $u.left$ and $u.right$. In case $u \prec u.left$ holds, u forwards $u.left$ by locally calling the action Linearize($u.left$) and then setting $u.left \leftarrow \perp$. Analogously, u performs the consistency check with its variable $u.right$. After the consistency checks, u introduces itself to its left and right neighbor by calling the action Linearize(u) on them.

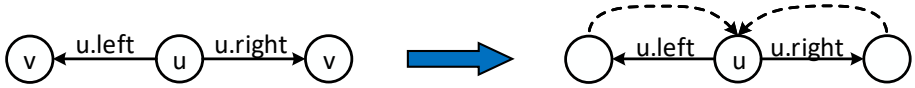
Upon executing Linearize(v) at u for some node $v \prec u$, u proceeds based on one of the following two cases:

(i) If $v \prec u.left$, then u just forwards v to $u.left$.

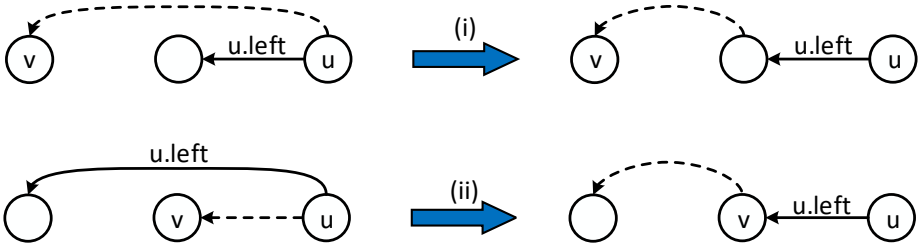
(ii) If $u.left \prec v \prec u$, then u has found a new closest left neighbor: Consequently, u forwards $u.left$ to v and stores v in $u.left$ afterwards.

In case $u \prec v$, u proceeds analogously using its variable $u.right$ in this case. Consider Figure 6.3 for an illustration of the Timeout and Linearize action.

From [ORS07] we derive the following theorem:



(a) Timeout executed at u.



(b) Linearize(v) executed at u for the cases $v \prec u.left$ and $u.left \prec v \prec u$.

Figure 6.3.: Illustration of the actions Timeout and Linearize (for u's left neighbor).

Theorem 6.12. *BuildList is self-stabilizing with regard to the legitimate state from Definition 6.11.*

Proof (Sketch). In order to show convergence, consider a pair of nodes (u, v) with $u \prec v$ that is adjacent in legitimate states. As the graph G is weakly connected, there exists an undirected path P from u and v. Let $P_{min}(u, v) = \text{argmin}_{w \in P}\, id(w)$ be the node with minimum identifier among all nodes on the path from u to v. Similarly, let $P_{max}(u, v) = \text{argmax}_{w \in P}\, id(w)$ be the node with maximum identifier among all nodes on the path from u to v. Define the potential function $\Phi(u, v) = P_{max}(u, v) - P_{min}(u, v)$. One can show via case distinction that Φ monotonically decreases over time until it holds that $\Phi = id(v) - id(u)$. This corresponds to (u, v) being directly connected, which implies convergence.

For closure, we argue that an explicit edge (u, v) is only forwarded, if u gets to know a closer neighboring node than v. However, this is not possible as the nodes already form a sorted list, so closure holds. □

6.5. Self-Stabilizing Sorted Rings

We extend BuildList to a (also well-known) protocol for a sorted ring, called BuildRing. To do so, each node u maintains an additional variable $u.ring \in V \cup \{\perp\}$. In BuildRing we distinguish between *list edges* (represented by $u.left$ and $u.right$) and *ring edges* (represented by $u.ring$).

Definition 6.13 (Legitimate State for BuildRing)**.** *The system is in a legitimate state for BuildRing if the following properties hold for each node $u \in V$:*

(a) *The explicit list edges represented by $u.left$ and $u.right$ are in a legitimate state according to Definition 6.11.*

(b) *Let $v_{min} \in V$ be the node that is minimal w.r.t. $\prec$ and let $v_{max} \in V$ be the node that is maximal w.r.t. $\prec$. If $u = v_{min}$, then $u.ring = v_{max}$. Similarly, if $u = v_{max}$, then $u.ring = v_{min}$. If $v_{min} \neq u \neq v_{max}$, then $u.ring = \perp$.*

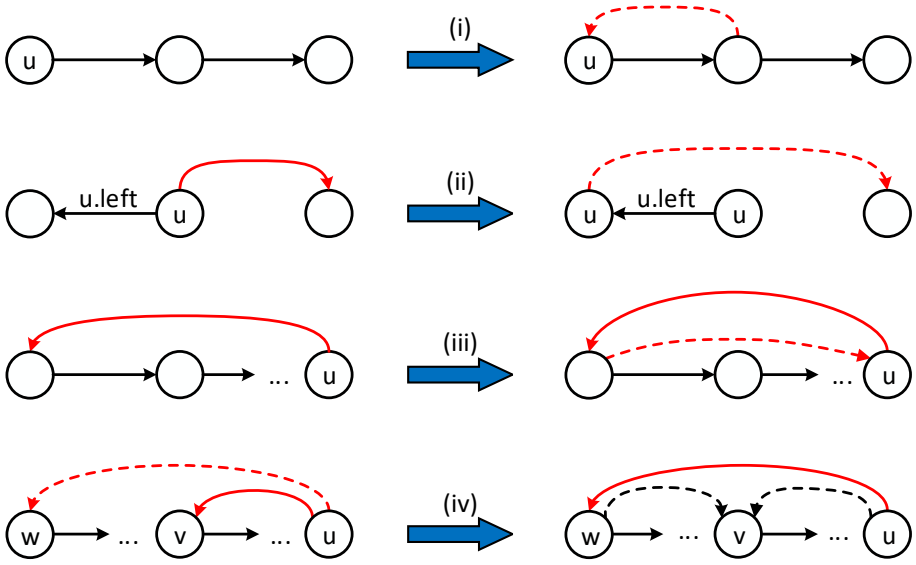


Figure 6.4.: Illustration of the additional actions of BuildRing. Red edges denote ring edges, black edges denote list edges.

List edges are handled by the BuildList protocol as usual. We introduce an extension to the Timeout action of BuildList and a new action IntroduceRing, the latter being responsible for forwarding ring edges. Intuitively, we let nodes u that do not have a left (or right) neighbor create new implicit ring edges to themselves in Timeout. These ring edges are then forwarded along the sorted line via IntroduceRing until they reach the node that is furthest away from u. We describe these extensions in more detail now for each node $u \in V$ (see also Figure 6.4 for an illustration):

(i) If in Timeout, u does not have a ring edge and does not have a left (or right) list neighbor, then u creates a ring edge to itself and forwards that edge to $u.left$ (or $u.right$) by calling the action IntroduceRing(u) on $u.right$ (or on $u.left$).

(ii) If $u.left \neq \perp$ (or $u.right \neq \prec$) and $u.ring \neq \perp$ with $u \prec u.ring$ (or $u.ring \prec u$) in Timeout, then u forwards $u.ring$ to $u.left$ (or $u.right$) by calling the action IntroduceRing($u.ring$) on $u.left$ (or on $u.right$).

(iii) If u has a ring edge that cannot be forwarded via the action (ii), then u introduces itself to $u.ring$ in Timeout by calling IntroduceRing(u) on $u.ring$, generating the implicit edge $(u.ring, u)$.

(iv) If u has an explicit ring edge (u, v) and receives an implicit ring edge to some node w via IntroduceRing(w), then u keeps the node stored in $u.ring$ that is further away from u according to $\prec$. The other edge, say (u, v), is then forwarded to the BuildList protocol; i.e., it is transformed into a list edge. u also introduces v to w via the Linearize action of the BuildList protocol.

Algorithm 10 The BuildRing Protocol (executed by each node $u \in V$)

```
1:  Timeout: true →
2:      if u.ring =⊥ then
3:          if u.left =⊥ ∧u.right ≠⊥ then
4:              u.right ← IntroduceRing(u)
5:          if u.left ≠⊥ ∧u.right =⊥ then
6:              u.left ← IntroduceRing(u)
7:      else
8:          if u.left ≠⊥ ∧u ≺ u.ring then
9:              u.left ← IntroduceRing(u.ring)
10:             u.ring ←⊥
11:         if u.right ≠⊥ ∧u.ring ≺ u then
12:             u.right ← IntroduceRing(u.ring)
13:             u.ring ←⊥
14:         if (u.left =⊥ ∧u ≺ u.ring) ∨ (u.right =⊥ ∧u.ring ≺ u) then
15:             u.ring ← IntroduceRing(u)
16:     Call Timeout from Algorithm 9

17: IntroduceRing(v):
18:     if u.ring =⊥ then
19:         if (v ≺ u ∧ u.right =⊥) ∨ (u ≺ v ∧ u.left =⊥) then
20:             u.ring ← v
21:         if v ≺ u ∧ u.right ≠⊥ then
22:             u.right ← IntroduceRing(v)
23:         if u ≺ v ∧ u.left ≠⊥ then
24:             u.left ← IntroduceRing(v)
25:     else
26:         if v ≺ u ∧ u.ring ≺ u then
27:             Let w ∈ {v, u.ring} be the node that is further away from u
28:             Let w′ ∈ {v, u.ring} be the other node
29:             u.ring ← w
30:             u ← Linearize(w′)              ▷ Action Linearize from Algorithm 9
31:             w ← Linearize(w′)
32:         else if (v ≺ u ≺ u.ring) ∨ (u.ring ≺ u ≺ v) then
33:             u ← Linearize(v)
34:             u ← Linearize(u.ring)
35:             u.ring ←⊥
```

From [KKS14] we derive the following theorem:

Theorem 6.14. *BuildRing is self-stabilizing with regard to the legitimate state from Definition 6.13.*

Proof (Sketch). Via case distinction, one can show that for each ring edge that connects two connected components, BuildRing eventually generates a corresponding list edge. Therefore, it follows that the subgraph of G induced by the list edges

eventually becomes weakly connected. Applying Theorem 6.12 implies that the sorted list converges. Now, once the nodes have already formed a sorted list, one can show that eventually all ring edges vanish, except the ring edges between the nodes v_{min}, v_{max} with minimum and maximum identifier (such that Definition 6.13(b) is satisfied). For this, notice that according to our rules, ring edges are always forwarded in the sorted list via the action IntroduceRing until they reach v_{min} (or v_{max}). Upon receiving an implicit ring edge e to v_{max}, the node v_{min} transforms e into an explicit ring edge by storing v_{max} in $v_{min}.ring$. In its Timeout action, v_{min} then introduces itself to v_{max} via the rule (iii), such that an explicit ring edge (v_{max}, v_{min}) is created by v_{max}. Therefore, BuildRing converges.

The closure property follows from the closure of BuildList (Theorem 6.12) and the fact that BuildRing does not create any additional explicit ring edges but only implicit edges (v_{min}, v_{max}), (v_{max}, v_{min}), which are immediately merged with their corresponding explicit edges. □

Whenever we want to make use of a self-stabilizing sorted ring, we use the BuildRing protocol. However, in order to provide a clean presentation of our algorithms, we omit the variable $u.ring$ and instead just assume that each node u has stored its left and right ring neighbors in the variables $u.left$ and $u.right$, respectively. This means that in legitimate states, the node v_{min} stores the node v_{max} in its variable $v_{min}.left$ and, analogously, v_{max} stores v_{min} in its variable $v_{max}.right$.

CHAPTER 7

Self-Stabilizing Generalized De Bruijn Graphs

In this first technical chapter of the second part, we investigate self-stabilizing overlay networks that are able to route messages with as few hops as possible. Such networks are useful for tasks such as in real-time applications like search engines, multiplayer games or social media networks, as the performance of these kinds of systems benefits from a low latency/delay. For example, experiments in [Bru09] show that users issue fewer search requests when the latency on Google web servers is increased by only 100 ms. For many systems there are hard deadlines on the delay that is acceptable: Multiplayer games often require server-side delays of at most 10 ms.

To keep the delay low, we require an overlay network to form a topology with a low diameter in legitimate states such that requests can be delivered quickly to the correct entity. We are interested in self-stabilizing systems that are able to route requests to their target as fast as possible even under a large number of participants. For example, routing in a simple line structure takes $\Theta(n)$ hops, whereas routing in a De Bruijn graph can be done in $\mathcal{O}(\log n)$ hops. Both of these structures have only a constant node degree. If the degree of the nodes is much higher (i.e., in a clique), routing can be done way more effectively. We can send requests to their destination in only one hop, since every node is connected to every other node in the system. The drawback here is that nodes have to maintain a large number of outgoing edges, which may be very costly w.r.t. storage space.

Our goal is to develop a self-stabilizing protocol for a network in which the node degree is lower than the node degree in the clique, but still enables to route requests to their destination in a constant number of hops w.h.p. Given a constant $d \geq 2$, our network has a diameter of d (w.h.p.) in every legitimate state. As a network topology, we use the generalization of the standard d-dimensional De Bruijn graph, called a *generalized De Bruijn* graph.

The self-stabilizing protocol consists of a combination of sub-protocols. We rely on a sorted list onto which we build additional connections, resulting in a sorted $\sqrt[d]{n}$-connected list. We use this topology to establish De Bruijn edges for $\log(\sqrt[d]{n})$ levels, such that each node has an outdegree of $\Theta(\sqrt[d]{n})$ in the resulting structure. By doing so, we achieve an asymptotically optimal balance between the degree of nodes and the diameter of the network.

Underlying Publication. This chapter is based on the following publication:

> M. Feldmann and C. Scheideler. "A Self-stabilizing General De Bruijn Graph". In: *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2017, cf. [FS17].

Outline of This Chapter. We first extend the classical definition for the De Bruijn graph to the generalized version (Section 7.1). Then we provide some additional related work in Section 7.2. We introduce the generalized De Bruijn network, the goal of stabilization, along with a constant-hop routing algorithm for that network in Section 7.3. Finally, in Section 7.4 we present and analyze our self-stabilizing protocol for the generalized De Bruijn network.

7.1. Generalized De Bruijn Graphs

Recall from Definition 2.5 that the *standard (d-dimensional) De Bruijn graph* consists of nodes with labels $(x_1, \ldots, x_d) \in \{0,1\}^d$ and edges $(x_1, \ldots, x_d) \to (j, x_1, \ldots, x_{d-1})$ for all $j \in \{0,1\}$. In order to get from any source node $s = (s_1, \ldots, s_d) \in \{0,1\}^d$ to any target node $t = (t_1, \ldots, t_d) \in \{0,1\}^d$, one can shift the bits of t one by one to the left of s, resulting in the routing path

$$((s_1, \ldots, s_d), (t_d, s_1, \ldots, s_{d-1}), (t_{d-1}, t_d, s_1, \ldots, s_{d-2}), \ldots, (t_1, \ldots, t_d)).$$

The d-dimensional De Bruijn graph therefore has a diameter of d. We call one single bit shift a *De Bruijn hop*.

If we assume d to be a constant, then the number of hops needed to route a message from any source node s to any target node t is constant. However, for a fixed value of d, the standard De Bruijn graph has a fixed number of nodes, that is, $n = 2^d$. Since we want to allow an arbitrary number of nodes in the system, the standard De Bruijn graph does not fit this purpose. Therefore, we extend the standard De Bruijn graph to the generalized De Bruijn graph by allowing nodes to use more digits for their labels than just from the set $\{0,1\}$.

Definition 7.1. *Let $q, d \in \mathbb{N}$. The* generalized (q-ary, d-dimensional) De Bruijn graph *consists of nodes with labels $(x_1, \ldots, x_d) \in \{0, \ldots, q-1\}^d$ and edges*

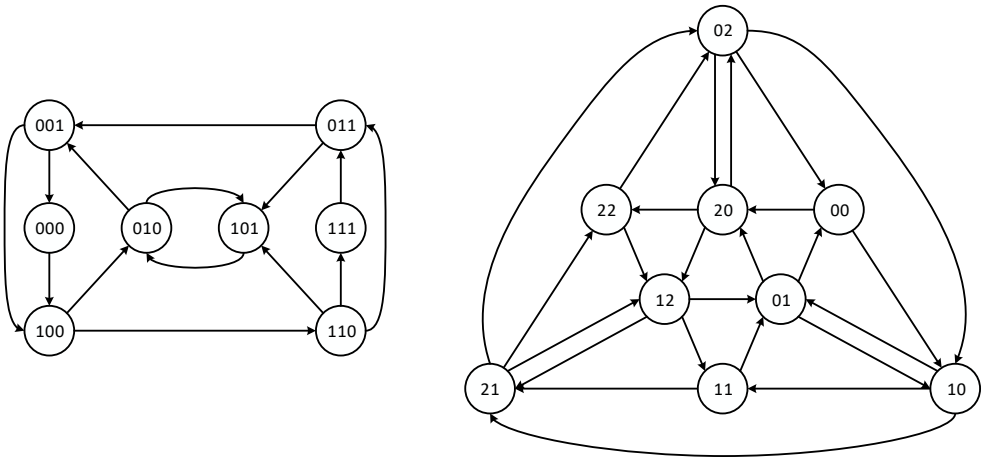
$$(x_1, \ldots, x_d) \to (j, x_1, \ldots, x_{d-1})$$

for all $j \in \{0, \ldots, q-1\}$.

Consider Figure 7.1 for an illustration of a standard De Bruijn graph and a generalized De Bruijn graph.

The diameter of the generalized De Bruijn graph is d, so we are still able to route search requests in d hops by performing exactly d bit shifts. Now we can fix d to some constant and obtain constant-hop routing paths for any message, while being able to allow an arbitrary number of nodes in the system by allowing the parameter q to be dynamic. In fact, it holds that $n = q^d$ and thus each node has a degree of $q = \sqrt[d]{n}$. Thus, the degree of the generalized De Bruijn graph is minimal with regard to its diameter.

Fact 7.2. *Every graph with n nodes and diameter d must have a degree of at least $\lfloor \sqrt[d]{n} \rfloor$.*



(a) Standard De Bruijn graph. (b) Generalized De Bruijn graph.

Figure 7.1.: Illustration of a standard De Bruijn graph ($n = 8, d = 3$) and a generalized De Bruijn graph ($n = 9, q = 3, d = 2$).

Proof. Assume to the contrary for a graph with n nodes and diameter d that no node has a degree higher than $\lfloor \sqrt[d]{n} \rfloor - 1$. Fix a node u and construct the BFS tree starting at u until level d. The number of leaf nodes in this tree is equal to $(\lfloor \sqrt[d]{n} \rfloor - 1)^d < (\lfloor \sqrt[d]{n} \rfloor)^d \leq n$, which implies that there exists a node that cannot be reached from u within exactly d hops, which is a contradiction. □

7.2. Related Work

In addition to Section 6.3 we give an overview on further related work specifically relevant to generalized De Bruijn graphs and overlay networks with a constant diameter.

Generalized De Bruijn Graphs. The standard De Bruijn graph is due to Nicolas G. De Bruijn [De 46] and its generalized version has been presented by Imase and Ito [II81] and by Reddy, Pradhan and Kuhl [RPK80]. Since then, several follow-up works have been published regarding graph-theoretical properties of the generalized De Bruijn graph [LZ91; Mau92; SSO94; GW94; KF12; OL13].

There already exists a self-stabilizing protocol by Richa et al. [RSS11] for the standard De Bruijn graph. The authors let each real node emulate 3 virtual nodes. By arranging those virtual nodes in a sorted list, one is able to emulate a De Bruijn hop by going from a real node to one of its virtual nodes and then searching in the sorted list for the next real node. Routing between any pair of nodes is then realized in $\mathcal{O}(\log n)$ hops w.h.p. by performing $\mathcal{O}(\log n)$ De Bruijn hops. Unfortunately, this approach cannot be trivially extended for generalized De Bruijn graphs as it would require each real node to emulate $\Theta(\sqrt[d]{n})$ virtual nodes. While we could potentially emulate a De Bruijn hop in the generalized De Bruijn graph by locally forwarding the packet from a real node to a virtual node, the number of hops that are needed in order to find the next real node in the sorted list grows up to $\mathcal{O}(\sqrt[d]{n})$.

Constant-Diameter Overlay Networks. Peer-to-peer overlays that are able to route requests to the target in one hop [GLR03; RLS02], two hops [GLR04] or a constant amount of hops [Gup+03] have already been proposed. Another protocol that provides fast but sometimes suboptimal routing as well as a handling of path outages, is the *Resilient Overlay Network* (RON) [And+02]. However, neither of the above protocols are truly self-stabilizing. There exists a self-stabilizing clique [KKS14] that provides routing within only one hop, but the node degree is n for each node in legitimate states, which may limit the scalability of the system when n gets large.

7.3. Network Topology and Routing

In this section we present the topology that we want nodes to form when being in a legitimate state. Afterwards, we present a routing algorithm for routing any message from any source node s to any target node t within a constant amounts of hops in this topology.

7.3.1. Network Topology

Given a set V of n nodes, we first hash the identifier $id(u)$ of each node $u \in V$ to the $[0,1)$-interval uniformly at random, using the pseudorandom hash function $h : \mathbb{N} \to [0,1)$. Define $\prec$ to be the total order on all nodes in V on the $[0,1)$-interval: i.e., for two nodes $u, v \in V$ it holds that $u \prec v$ if and only if $h(id(u)) < h(id(v))$. If clear from the context, we just use u or $h(u)$ instead of $h(id(u))$. Recall Definition 6.9 for notation on left, right, closest left and closest right neighbors. We introduce some additional notation here:

Definition 7.3. *Let $u, v, w \in V$. We say that u* is closer to v than w *if and only if $|u - v| < |v - w|$. A node u is* closest to some point $p \in [0,1)$ *if and only if $|u - p| < |v - p|$ for all nodes $v \neq u$.*

The network we introduce in the following definition has a diameter of d w.h.p., which makes routing possible in a constant number of hops:

Definition 7.4 (Network Topology)**.** *Let V be a set of n nodes, $d \geq 2, \zeta \geq 4$ be fixed constants, $q = \sqrt[d]{n}$ and $\prec$ be a total order on all nodes in V. The* general De Bruijn network (GDB) *is a directed graph $G = (V, E_L \cup E_Q \cup E_{DB})$ with the following properties:*

(a) $(u, v) \in E_L \Leftrightarrow u$ *is the closest left neighbor of* v.

(b) $(u, v) \in E_Q \Leftrightarrow v \in \left[u - \frac{\zeta}{2} \cdot \frac{q}{n}, u + \frac{\zeta}{2} \cdot \frac{q}{n}\right]$

(c) $\forall i \in \{1, \ldots, \log(q)\}\ \forall j \in \{0, \ldots, 2^i - 1\} : (u, v) \in E_{DB} \Leftrightarrow v$ *is closest to the point* $\frac{u+j}{2^i}$.

Call edges in E_L list edges*, edges in E_Q* q-neighborhood edges *and edges in E_{DB}* De Bruijn edges.

For an edge $(u, v) \in E_{DB}$ where v is closest to the point $\frac{u+j}{2^i}$, denote the edge (u, v) as a *De Bruijn edge on level* i, $i \in \{1, \ldots, \log(q)\}$. In case $i = 1$, such an edge resembles a standard De Bruijn edge from Definition 2.5; for $i = \log q$, the edge resembles a generalized De Bruijn edge from Definition 7.1. Note that we include De Bruijn edges on levels $2, \ldots, \log q - 1$ to facilitate the self-stabilization process. If we forward a message via a De Bruijn edge on level i we denote this as a *De Bruijn hop*. For $i > 1$, we denote this as a *general De Bruijn hop* and for $i = 1$, we denote this as a *standard De Bruijn hop*. By writing $u \to p$ for a node $u \in V$ and a point $p \in [0, 1)$, we mean that u has an edge to the node $v \in V$ that is closest to p; i.e., u stores the reference of v in its local memory. We define the labels of the nodes such that they correspond to the node labels of the standard De Bruijn graph (see Definition 2.5).

Definition 7.5. *Let $u \in V$ with $h(id(u)) \in [0, 1)$. Define the* label *of u by $label(u) = (x_1, \ldots, x_m) \in \{0, 1\}^m$ such that*

$$h(id(u)) = \sum_{k=1}^{m} x_k \cdot \frac{1}{2^k}.$$

We are now ready to show that the edges E_{DB} in the GDB emulate the edges of the standard De Bruijn graph and the generalized De Bruijn graph correctly. Here we assume for simplicity that for a De Bruijn hop via $u \to \frac{u+j}{2^i}$ there exists a node v with $h(id(v)) = \frac{u+j}{2^i}$. As it will turn out, having no node with a hash value of exactly $\frac{u+j}{2^i}$ is no problem when constructing a d-hop routing algorithm for the GDB.

Lemma 7.6. *Let $u \in V$ and fix values $i \in \{1, \ldots, \log(q)\}$, $j \in \{0, \ldots, 2^i - 1\}$. A De Bruijn hop via $u \to \frac{u+j}{2^i}$ is equivalent to appending i bits $y_{i-1}, y_{i-2}, \ldots, y_0 \in \{0, 1\}^i$ to the left of $label(u)$. For the appended bit string $(y_{i-1}, y_{i-2}, \ldots, y_0) \in \{0, 1\}^i$ it holds that*

$$y_{i-1} \cdot 2^{i-1} + y_{i-2} \cdot 2^{i-2} + \ldots + y_0 \cdot 2^0 = j.$$

Proof. W.l.o.g. assume that there exists a node v with $h(id(v)) = \frac{u+j}{2^i}$. We have

$$\begin{aligned}
\frac{u+j}{2^i} &= \frac{u}{2^i} + \frac{j}{2^i} \\
&= \frac{\sum_{k=1}^{m} x_k \cdot \frac{1}{2^k}}{2^i} + \frac{j}{2^i} \\
&= \sum_{k=1}^{m} x_k \cdot \frac{1}{2^{k+i}} + \frac{j}{2^i} \\
&= \underbrace{\frac{x_1}{2^{1+i}} + \frac{x_2}{2^{2+i}} + \ldots + \frac{x_m}{2^{m+i}}}_{\text{Bits of } label(u) \text{ shifted } i \text{ positions to the right}} + \frac{j}{2^i}.
\end{aligned}$$

We know that $j \in \{0, \ldots, 2^i - 1\}$, so we can write j as a binary string $(y_{i-1}, \ldots, y_0) \in \{0, 1\}^i$ with

$$j = \sum_{k=0}^{i-1} y_k \cdot 2^k = y_{i-1} \cdot 2^{i-1} + y_{i-2} \cdot 2^{i-2} + \ldots + y_0 \cdot 2^0.$$

Plugging the representation for j into the equation above we get

$$\begin{aligned}
\frac{v+j}{2^i} &= \frac{j}{2^i} + \sum_{k=1}^{m} x_k \cdot \frac{1}{2^{k+i}} \\
&= \frac{\sum_{k=0}^{i-1} y_k \cdot 2^k}{2^i} + \sum_{k=1}^{m} x_k \cdot \frac{1}{2^{k+i}} \\
&= \sum_{k=0}^{i-1} y_k \cdot 2^{k-i} + \sum_{k=1}^{m} x_k \cdot \frac{1}{2^{k+i}} \\
&= \underbrace{\frac{y_{i-1}}{2} + \frac{y_{i-2}}{4} + \ldots + \frac{y_0}{2^i}}_{i \text{ bits defined by } j \text{ appended to the left}} + \underbrace{\frac{x_1}{2^{i+1}} + \frac{x_2}{2^{i+2}} + \ldots + \frac{x_m}{2^{i+m}}}_{\text{Bits of } label(u) \text{ shifted } i \text{ positions to the right}}.
\end{aligned}$$

Therefore, we have $label(v) = (y_{i-1}, y_{i-2}, \ldots, y_0, x_1, \ldots, x_m)$, which corresponds to appending i bits $y_{i-1}, y_{i-2}, \ldots, y_0$ to the left of $label(u)$. This proves the lemma. □

Since $j \in \{0, \ldots, 2^i - 1\}$, we are able to append any arbitrary bit string of length i to the label of any node u via a De Bruijn hop. So for $i = \log(q)$, we are able to append $\log(q) = \log(\sqrt[d]{n}) = \frac{1}{d}\log(n)$ arbitrary bits at once per general De Bruijn hop.

7.3.2. Routing

Now we present a routing algorithm DBSearch that routes a message from any source node $s \in V$ to any target node $t \in V$ within d hops in the GDB. In order to provide a clean presentation of the routing algorithm, we assume that the label of each node consists of at least $\log n$ bits.

DBSearch (Algorithm 11) proceeds in two phases. In the first phase, we perform $d-1$ general De Bruijn hops to append the most significant $\lceil \frac{d-1}{d} \log n \rceil$ bits of $label(t)$ to the left of $label(s)$. In the second phase, we greedily search for t via q-neighborhood edges.

At the source node s, DBSearch is initialized with the identifier $id(t)$ of the target node t and a variable $i = d - 1$ that counts the remaining number of De Bruijn hops to be executed in the first phase. DBSearch determines for each node u on the routing path to t the next node v on that path and forwards the message to v. Once the message has arrived at t, the algorithm outputs *Success* if there does not exist any node with identifier $id(t)$.

Phase 1: General De Bruijn Hops. The first phase consists of $d - 1$ general De Bruijn hops. At the beginning of the first phase, we compute the label $label(t)$ of t. Consider the most significant $m = \lceil \frac{d-1}{d} \log n \rceil$ bits $t_1, \ldots, t_m$ of $label(t)$. For the i-th general De Bruijn hop we consider the bits $t_{(i-1)\cdot\log q+1}, \ldots t_{i\cdot\log q}$ and compute

$$j = \sum_{k=(i-1)\cdot\log q+1}^{i\cdot\log q} t_k \cdot 2^k.$$

Observe that $j \in \{0, \ldots, q-1\}$. We then perform a general De Bruijn hop via the edge $u \to \frac{u+j}{q}$.

Algorithm 11 The routing algorithm DBSearch, executed by node $u \in V$

```
1:  DBSearch(id(t), i)
2:      if id(u) = id(t) then
3:          output Success
4:      if i > 0 then
5:          Let label(t) = (t_1, ..., t_m) ∈ {0,1}^m
6:          Compute j = Σ_{k=(i-1)·log q+1}^{i·log q} t_k · 2^k
7:          w ← argmin_{v∈V,(u,v)∈E_DB} |v − (u+j)/q|
8:          w ←DBSearch(id(t), i − 1)
9:      else
10:         w ← argmin_{v∈V,(u,v)∈E_Q} |v − h(id(t))|
11:         if |w − h(id(t))| < |u − h(id(t))| then
12:             w ←DBSearch(id(t), 0)
13:         else
14:             output Failure
```

Phase 2: Greedy Search. In the second phase, we greedily search for the target node t by delegating the message via edges in E_q. We do this until t has been found, or until the message arrives at a node $v \in V, id(v) \neq id(t)$ from which it cannot be forwarded to a node that is closer to $h(id(t))$ than v. In both cases, the algorithm terminates, resulting in a successful delivery of the message in the first case or a failed delivery in the second case since no node with ID $id(t)$ exists in the system. This phase is equivalent to fixing the remaining bits of $label(t)$, which can be done via a single hop until the request arrives at the target node.

We rely on [NW07, Lemma 4.1] for parts of the analysis. The lemma gives upper and lower bounds for the the distance between to consecutive nodes in the sorted list and is stated in the following:

Lemma 7.7 ([NW07])**.** *After inserting n random points on the $[0,1)$-interval the length of the longest segment is w.h.p $\Theta\left(\frac{\log n}{n}\right)$. With high probability there is no segment which is shorter than $\Theta\left(\frac{1}{n^2}\right)$.*

Lemma 7.7 implies that a single De Bruijn hop imposes an additive error of at most $\Theta\left(\frac{\log n}{n}\right)$ w.h.p., i.e., the node w that is closest to the point $p = \frac{v+j}{q}$ is at most $\Theta\left(\frac{\log n}{n}\right)$ units away from p.

The following theorem yields the desired bound on the number of hops for DBSearch:

Theorem 7.8. *The number of hops required to send a message from a source node $s \in V$ to a target node $t \in V$ via DBSearch is d w.h.p.*

Proof. Let $label(t) = (t_1, \ldots, t_m) \in \{0,1\}^m$ be the label of t. In the first phase of DBSearch, we perform $d-1$ general De Bruijn hops. Lemma 7.6 implies that we arrive at some node v with the label $(v_1, \ldots, v_k)$, and

$$v_i = t_i \; \forall \; i \in \left\{1, \ldots, \left\lceil \frac{d-1}{d} \log(n) \right\rceil \right\}.$$

Assume $m = k$ for convenience. At this point, $i = 0$, so DBSearch switches to Phase 2. The remaining bits that need to be fixed are the bits $t_{\lceil \frac{d-1}{d} \log(n) \rceil + 1}, \ldots, t_m$, keeping an additive error of at most $(d-1)\frac{c \log n}{n}$ w.h.p. in mind for some constant $c > 0$ (Lemma 7.7). We show that these bits can be fixed in one single hop via the q-neighborhood, because in the worst case it holds that

$$v_i \neq t_i \ \forall \ i \in \left\{ \left\lceil \frac{d-1}{d} \log(n) \right\rceil + 1, \ldots, m \right\},$$

so the maximum distance between $h(id(v))$ and $h(id(t))$ on the $[0,1)$-interval is equal to

$$\begin{aligned}
(d-1)\frac{c \log n}{n} + \sum_{i=\lceil \frac{d-1}{d} \log(n) \rceil + 1}^{k} \frac{1}{2^i} &\leq (d-1)\frac{c \log n}{n} + \underbrace{\left(\sum_{i=1}^{k} \frac{1}{2^i} \right)}_{\leq 1 \text{ for } k \to \infty} \cdot \frac{1}{n^{\frac{d-1}{d}}} \\
&\leq (d-1)\frac{c \log n}{n} + \frac{1}{n^{\frac{d-1}{d}}} \\
&= (d-1)\frac{c \log n}{n} + \frac{\sqrt[d]{n}}{n} \\
&= (d-1)\frac{c \log n}{n} + \frac{q}{n} \\
&< \frac{\zeta}{2} \cdot \frac{q}{n}
\end{aligned}$$

for $\zeta \geq 4$ and n high enough. Therefore, it holds that $(v,t) \in E_Q$, so we are able directly go from v to t via one single hop. □

Notice that Theorem 7.8 still holds when q is not exactly accurate but only a value in $\Theta(\sqrt[d]{n})$ that is at least as large as $\sqrt[d]{n}$. This is important because the self-stabilizing protocol presented in the next section uses an approximation of q with the above properties.

7.4. Protocol BuildGDB

In this section we describe our self-stabilizing protocol for the generalized De Bruijn graph, called BuildGDB. We construct the protocol from sub-protocols for each type of edges mentioned in Definition 7.4.

7.4.1. Protocol Description

We first give an overview of the variables of each node. Throughout the rest of this chapter, we assume that all nodes are aware of the constants $d \geq 2$ and $\zeta \geq 8$ (for technical reasons that will become clear in the analysis, we choose $\zeta \geq 8$ as opposed to $\zeta \geq 4$ from Definition 7.4).

Definition 7.9. *For the BuildGDB protocol, each node $u \in V$ maintains the following variables:*

(a) *Variables* $u.left, u.right \in V \cup \{\perp\}$ *storing* u*'s left and right list neighbor.*

(b) *Variables* $u.q_l, u.q_r \in \left\{ \frac{\zeta}{2^k} \mid k \in \mathbb{N}_0 \right\}$ *storing estimates of* $\frac{\zeta \sqrt[d]{n}}{4n}$.

(c) *A set* $u.Q_l \subset V$ *storing all nodes in the interval* $[u - 2u.q_l]$ *and a set* $u.Q_r \subset V$ *storing all nodes in the interval* $[u, u + 2u.q_l]$.

(d) *Variables* $u.db(i,j) \in V \cup \{\perp\}$*, for all* $i \in \{1, \ldots, \lceil \log(\sqrt[d]{n}) \rceil\}$, $j \in \{0, \ldots, 2^i - 1\}$ *representing* u*'s De Bruijn edges.* $u.db(i,j)$ *stores the node that is closest to the point* $\frac{u+j}{2^i}$*. Denote the union of* u*'s De Bruijn edges by the set* $u.db = \bigcup_{i,j} u.db(i,j)$.

Observe that $u.db(1,0)$ and $u.db(1,1)$ represent u's standard De Bruijn edges. If BuildGDB has to call an action on a node stored in variable u, it executes this call only if $u \neq \perp$. BuildGDB consists of three sub-protocols: one for list edges, one for q-neighborhood edges, and one for De Bruijn edges. Each of these sub-protocols comes with a dedicated Timeout action as well as other protocol-specific actions. We describe each sub-protocol individually in the following subsections.

List Edges

The base of our self-stabilizing protocol consists of a sorted list for all nodes $u \in V$. For this we define the total order $\prec$ as already mentioned in Section 7.3. For two nodes $u, v \in V$ it holds that $u \prec v$ if and only if $h(id(u)) < h(id(v))$. We use the BuildList protocol from Section 6.4, using the variables $u.left$ and $u.right$.

Unfortunately, we cannot simply apply Theorem 6.12 to guarantee convergence for the sorted list in BuildGDB because we only require $G = (V, E_L \cup E_Q \cup E_{DB})$ to be weakly connected. Therefore, we *downgrade* (non-list) edges represented by sets $u.Q_l$, $u.Q_r$ and $u.db$. Downgrading some node $v \in u.Q_l \cup u.Q_r \cup u.db$ at node u is done in a round-robin fashion in the Timeout action of each sub-protocol other than BuildList, by locally calling Linearize(v). This creates an implicit list edge (u, v).

Q-Neighborhood

Next, we describe how nodes are able to establish and maintain connections to their closest $\Theta(\sqrt[d]{n})$ neighbors (called *q-neighborhood* for $q = \sqrt[d]{n}$) along with an approximation for the value $\Theta(\sqrt[d]{n})$. Algorithm 12 states the pseudocode for the q-neighborhood protocol.

Every node $u \in V$ aims to keep edges to all nodes in the interval $[u - \frac{\zeta}{2}\sqrt[d]{n}, u + \frac{\zeta}{2}\sqrt[d]{n}]$. Since u is not able to determine the exact value of $\sqrt[d]{n}$ locally, it stores an approximation of $\frac{\zeta \sqrt[d]{n}}{4n}$ in its variables $u.q_l, u.q_r$ for its left and right side, respectively, and aims to establish connections in $u.Q_l, u.Q_r$ to all nodes that are contained in the interval $[u - 2u.q_l, u + 2u.q_r]$. As it will turn out, this results in $|u.Q_L| + |u.Q_r| \in \Theta(\sqrt[d]{n})$ w.h.p. in legitimate states. Next, we describe how BuildGDB updates $u.Q$ and how $\frac{\zeta \sqrt[d]{n}}{4n}$ is approximated.

Maintaining $u.Q_l$ and $u.Q_r$. We just describe the algorithm for maintaining the sets $u.Q_l$ and $u.Q_r$ for the set $u.Q_r$ as the algorithm for $u.Q_l$ works analogously. Assume

Algorithm 12 The q-neighborhood sub-protocol of BuildGDB, executed by node u

```
1:  Timeout → true
2:      Remove and downgrade nodes v ∈ u.Q_l ∪ u.Q_r if v ∉ [u − 2u.q_l, u + 2u.q_r]
3:      if u.left ∉ u.Q_l ∧ u.left ∈ [u − 2u.q_l, u] then
4:          u.Q_l ← u.Q_l ∪ u.left
5:      if u.right ∉ u.Q_r ∧ u.right ∈ [u, u + 2u.q_l] then
6:          u.Q_r ← u.Q_r ∪ u.right
7:      Pick v_i ∈ u.Q_l ∪ u.Q_r in a round-robin fashion
8:      Linearize(v_i)
9:      if v_i ≠ u.left ∧ v_i ≠ u.right then
10:         v_i ← Introduce(v_{i−1}, u)
11:     else
12:         v_i ← Introduce(u, u)
13:     EstimateSqrtN()

14: Introduce(v, s)
15:     if v ∈ [u − 2u.q_l, u] then
16:         u.Q_l ← u.Q_l ∪ v
17:     else if v ∈ [u, u + 2u.q_r] then
18:         u.Q_r ← u.Q_r ∪ v
19:     else
20:         Linearize(v)
21:     if s ≠⊥ then
22:         if s ≺ u then
23:             s ← Introduce(u.right ⊥)
24:         else
25:             s ← Introduce(u.left, ⊥)
```

that $u.Q_r = \{v_1, \ldots, v_k\}$ with $v_i \prec v_{i+1}$ for $i = 1, \ldots, k-1$. To keep $u.Q_r$ updated, the node u does the following: In each call of Timeout u first picks $v_i \in u.Q_r$ in a round-robin fashion[1] and then introduces v_r to its closest list neighbor $\tilde{v} \in u.Q_r$ in the direction of u by calling the action Introduce($\tilde{v}$, u) on v_i. The node $\tilde{v}$ is determined as follows: If $v_i = u.right$, then $\tilde{v} = u$. Otherwise, $\tilde{v} = v_{i-1}$.

When some node u executes the action Introduce(v, s) for $v \in [u, u + 2u.q_r]$, u includes v into $u.Q_r$, otherwise v is forwarded via the BuildList protocol by locally calling Linearize(v) on u. Afterwards, u responds to the sender s of the Introduce(v, s) message by sending an Introduce($u.right$, $\perp$) message to s. By doing so we enable s to expand its set $s.Q_r$ in case it does not yet contain all nodes within the interval $[s, s + 2s.q_r]$. Note that the second parameter of the Introduce action is set to $\perp$ for this response, in order to avoid an infinite loop of message calls between two nodes.

[1] Picking a node v from some set S stored at node u in *round-robin fashion* can easily be realized in a self-stabilizing manner, by ordering the nodes in S arbitrarily and then picking the i-th node in that order. Here, i is an integer-variable that is updated by $i \leftarrow i + 1 \mod |S|$ whenever a node has been picked. This guarantees that when picking a node $|S|$ times, each node in S has been picked once.

Figure 7.2 illustrates these introduction rules for the q-neighborhood edges of a node u.

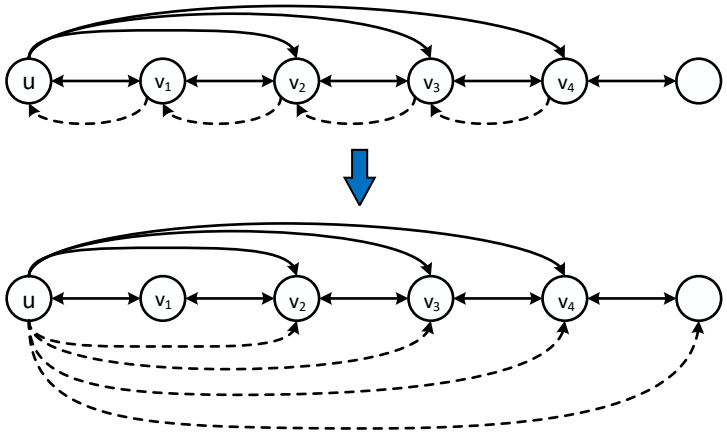


Figure 7.2.: Illustration of the introduction rules for the q-neighborhood sub-protocol. In the upper image u introduces itself to its list neighbor v_1 and also introduces v_1 to v_2 (which is v_2's closest list neighbor lying in the direction of u) as well as v_2 to v_3 and v_3 to v_4. Sending those introduction messages generates the corresponding (dashed) implicit edges shown in the upper image. Once u has received the responses from $v_1, \ldots, v_4$, the implicit edges shown in the bottom image are created.

Maintaining $u.q_l$ and $u.q_r$. Algorithm 13 states the pseudocode for the action EstimateSqrtN that is periodically executed by each node u at the end of its Timeout action. The algorithm and the description that follows show how to update the variable $u.q_r$. The procedure for updating $u.q_l$ works analogously. Recall that we aim for $u.q_r = \frac{\zeta \sqrt[d]{n}}{4n}$.

Algorithm 13 The action EstimateSqrtN, called locally by a node $u \in V$

```
1: EstimateSqrtN()
2:     Let I_r(x) = [u + ζ/2^x].
3:     Let N_r(x) = |{v ∈ u.Q_r | v ∈ I_r(x)}|.
4:     Let f_r(x) = (4/ζ)^d · |I_r(x)| − (1/N_r(x))^{d−1}.
5:     Compute x ∈ ℕ_0 s.t. f_r(x) is closest to 0 and x < N_r(x)/ζ − log N_r(x).
6:     u.q_r ← 1/2^x
```

Let $I_r(x) = [u + \frac{\zeta}{2^x}]$ be an interval of size $1/2^x$ and let $N_r(x)$ be the set of all nodes that are contained in $I_r(x)$. We seek to compute a value x for which the function

$$f_r(x) = \left(\frac{4}{\zeta}\right)^d \cdot |I_r(x)| - \left(\frac{1}{N_r(x)}\right)^{d-1}$$

is closest to 0. Here $f_r(x)$ sets the interval $I_r(x)$ into relation to $N_r(x)$. As it will turn out in the analysis, it holds w.h.p. that $f_r(x)$ is closest to 0 if $1/2^x$ is equal to

$\frac{\zeta \sqrt[d]{n}}{4n}$, as in this case $|I_r(x)| = \frac{\zeta \sqrt[d]{n}}{4n}$ and $N_r(x) \in \left(\frac{\zeta \sqrt[d]{n}}{8n}, \frac{\zeta \sqrt[d]{n}}{2n}\right)$. Setting $u.q_r$ to $1/2^x$ therefore eventually yields the desired estimate.

For technical reasons, we also have to make sure that $u.q_r \geq \frac{\zeta \log n}{n}$. For the computed value x we thus also require that $x < N_r(x)/\zeta - \log N_r(x)$ holds. The following lemma from [RSS11, Lemma 1] implies that in case $x < N_r(x)/\zeta - \log N_r(x)$, we get that the corresponding interval $I_r(x)$ and thus also $u.q_r$ have the size $\frac{\zeta \log n}{n}$.

Lemma 7.10 ([RSS11])**.** *Let $I(j) \subseteq (0,1)$ be any interval of size $(1/2)^j$ starting at a node u and let $N(j)$ be the number of nodes in $I(j)$. For any constant $c > 1$ there is a constant $\varepsilon \in (0,1)$ (that can be arbitrarily small depending on c) so that w.h.p. the following holds: if $|I(j)| < (1-\varepsilon)(c \log n/n)$ then $j > N(j)/c - \log N(j)$ and if $|I(j)| > (1+\varepsilon)(c \log n)/n$ then $j < N(j)/c - \log N(j)$.*

Finally, we want to note that once $u.q_r \in \Theta\left(\frac{\sqrt[d]{n}}{n}\right)$ is stable, the node u can compute $u.q \in \Theta(\sqrt[d]{n})$ with $u.q \geq \sqrt[d]{n}$: u just has to count the number of nodes $N_r(2u.q_r)$ in $[u, 2u.q_r]$ (to which it eventually will be connected via the q-neighborhood protocol). As it will turn out in the analysis, we have that $N_r(2u.q_r) \in \left(\frac{\sqrt[d]{n}}{8n}, \frac{\sqrt[d]{n}}{n}\right)$ w.h.p., so in order to get to know $u.q \in \Theta(\sqrt[d]{n})$ with $u.q \geq \sqrt[d]{n}$, u just computes $8 \cdot N_r(2u.q_r)$. Such an approximation is needed in order to establish the De Bruijn edges for each level $i \in \{1, \ldots, \lceil \log(\sqrt[d]{n}) \rceil\}$ (recall Definition 7.4) and to perform the routing algorithm from the previous section.

De Bruijn Edges

In this section we describe how the nodes are able to generate their correct De Bruijn edges in a self-stabilizing manner. We assume that each node u stores an approximation of $\Theta(\sqrt[d]{n})$ in the variable $u.q = 2^k$, $k \in \mathbb{N}_0$. Note that such a variable is not necessary technically because u is able to compute $\Theta(\sqrt[d]{n})$ locally at any time as explained in the previous section. Recall that each node u maintains a variable $u.db(i,j)$ ($i = \{1, \ldots, \log(u.q)\}$, $j = \{0, \ldots, 2^i - 1\}$) that has to store the node v closest to the point $\frac{u+j}{2^i}$. In case $u.q$ changes (i.e., it is either doubled or divided by some power of two) $u.db$ is updated accordingly. For the case that $u.q$ is reduced, u downgrades all node references that are not part of $u.db$ anymore to the BuildList protocol.

To establish De Bruijn edges for each level i we use a probing approach similar to the one presented in [RSS11]. In each call of Timeout, we pick $i \in \{1, \ldots, \log(u.q)\}$ and $j \in \{0, \ldots, 2^i - 1\}$ in a round-robin fashion. Node u first downgrades its node reference $u.db(i,j)$ to the BuildList protocol and then generates a message $M(i,j)$ (called *probe*) with target point $t = \frac{u+j}{2^i} \in [0,1)$ that aims to find the node v closest to t. We store i and j in $M(i,j)$ since these are important for routing $M(i,j)$ to its target location effectively. Once $M(i,j)$ has arrived at the node v that is closest to t, $M(i,j)$ is sent back to u storing the reference of v. We also keep the reference of u stored in $M(i,j)$ in order to be able to immediately return $M(i,j)$ to u from v. At the time u receives $M(i,j)$ from v an implicit edge (u,v) is generated, so u can then store v into $u.db(i,j)$. Depending on the values i and j, we use the following approach to route $M(i,j)$ to the target point t. At u we forward $M(i,j)$

to the node $v = u.db(i-1, k)$, with $k = j \mod 2^{i-1}$. In case $i = 0$, we forward $M(i, j)$ to $v = u.left$ (if $j = 0$) or to $v = u.right$ (if $j = 1$) instead. Next, at v we execute a standard De Bruijn hop: If $j \geq 2^{i-1}$ we forward $M(i, j)$ from v to $v.db(1, 1)$, otherwise we forward $M(i, j)$ from v to $v.db(1, 0)$. Now we greedily forward $M(i, j)$ via the q-neighborhood until some node w is reached that is closest to $\frac{u+j}{2^i}$ based on its local view. At w we store w's reference in $M(i, j)$ and send $M(i, j)$ back to u, such that u is able to set $u.db(i, j) = w$.

Note that if the system has not reached a legitimate state yet, the first two steps may not be executable, since the respective variables are set to $\perp$. In case the first step cannot be executed, we do not forward the probe at all. Once the sorted list is in a legitimate state, the first step can be executed for the probes $M(1, 0)$ and $M(1, 1)$. The set of De Bruijn edges is then built from the bottom levels up to the top levels, as one can easily show via induction. In case the second step cannot be executed we just proceed with the third step. Once the q-neighborhood has stabilized, $M(i, j)$ will eventually arrive at the correct target node. If node u updates $u.db(i, j)$, it forwards the old value for $u.db(i, j)$ to the BuildList protocol. Algorithm 14 states the pseudocode for the protocol.

Having nodes store lower-level De Bruijn edges is not only useful in our probing approach, but also reduces the effort for a node u when $u.q$ is reduced. This will certainly be the case in a dynamic environment as there are nodes leaving the system. As soon as $u.q$ is updated to $u.q/2^k$ for some $k \in \mathbb{N}$, u just downgrades and removes its De Bruijn edges on the k highest levels, which it can do locally. Without lower-level De Bruijn edges u would have to probe for a new set of De Bruijn edges from the ground up. Similarly, in the case that $u.q$ doubles, u is able to use its old De Bruijn edges on the highest level to effectively probe for the De Bruijn edges on the next higher level.

7.4.2. Analysis

In this section we first show that the BuildGDB protocol is self-stabilizing, meaning that BuildGDB satisfies the convergence and closure properties (Definition 6.5) w.r.t. legitimate state indicated by the general De Bruijn network (Definition 7.4). Afterwards, we show some additional interesting properties of our network.

Given any weakly connected graph $G = (V, E_L \cup E_Q \cup E_{DB})$, we first argue that all corrupted messages initially stored in node channels are processed, so the system arrives at a state where no corrupted messages exist:

Lemma 7.11. *Given any weakly connected graph $G = (V, E_L \cup E_Q \cup E_{DB})$ and a set of corrupted messages M that is spread arbitrarily over all node channels. Eventually, G is free of corrupted messages, while staying weakly connected.*

Proof. Any message $m \in M$ is processed by BuildGDB according to the (sub-)protocol descriptions from Section 7.4. By definition of BuildGDB, G does not get disconnected when processing m since we never just remove node references but instead either forward them to other nodes or downgrade them to the BuildList protocol. Also note that a corrupted message $m \in M$ may trigger the generation of a chain of further corrupted messages. However, this chain is finite as the implicit edge generated by m

Algorithm 14 The De Bruijn sub-protocol → executed by node u

1: Timeout → *true*
2: Remove and downgrade De Bruijn edges on levels $l > \log(u.q)$
3: Pick $i \in \{1, \ldots, \log(u.q)\}$ and $j \in \{0, \ldots, 2^i - 1\}$ in a round-robin fashion
4: Linearize($u.db(i, j)$)
5: **if** $i > 1$ **then**
6: $u.db(i-1, j \mod 2^{i-1}) \leftarrow$Probe($u$, $\frac{u+j}{2^i}$, i, j, *true*)
7: **else if** $j = 0$ **then** ▷ $i = 1, j = 0$
8: $u.left \leftarrow$Probe(u, $\frac{u+j}{2^i}$, i, j, *true*)
9: **else** ▷ $i = 1, j = 1$
10: $u.right \leftarrow$Probe(u, $\frac{u+j}{2^i}$, i, j, *true*)

11: Probe(s, t, i, j, *flag*)
12: **if** *flag* = *true* **then**
13: **if** $(u.db(1,0) = \perp \wedge j < 2^{i-1}) \vee (u.db(1,1) = \perp \wedge j \geq 2^{i-1})$**then**
14: Probe(s, t, i, j, *false*)
15: **if** $j < 2^{i-1}$ **then**
16: $u.db(1,0) \leftarrow$Probe(s, t, i, j, 0)
17: **else**
18: $u.db(1,1) \leftarrow$Probe(s, t, i, j, 0)
19: **else**
20: $v \leftarrow \text{argmin}_{w \in u.Q_l \cup u.Q_r \cup \{u\}} |w - t|$
21: **if** $u = v$ **then**
22: $s \leftarrow$ ProbeDone(u, i, j)
23: **else**
24: $v \leftarrow$Probe(s, t, i, j, 0)

25: ProbeDone(v, i, j)
26: **if** $v \neq u.db(i,j) \wedge u.db(i,j) \neq \perp$ **then**
27: Linearize($u.db(i, j)$)
28: $u.db(i,j) \leftarrow v$

is only forwarded for a finite amount of time until it is merged with or transformed into an explicit edge.

Messages m that are not of the form of any of the actions described in the protocol are immediately detected to be corrupted by the receiving node u. Node u then just forwards all node identifiers contained in m via the BuildList protocol and drops m afterwards. By doing so, m vanishes from the system.

The system therefore eventually arrives at a state where no more corrupted messages are contained in the node channels. □

For the rest of the analysis we assume that there are no corrupted messages in the system anymore. We show the convergence property in multiple phases: First we show that our system converges to a sorted list from any weakly connected graph. Once the sorted list is in a legitimate state, our protocol is able to establish the

q-neighborhood edges. After the q-neighborhood edges are established, BuildGDB eventually generates the correct De Bruijn edges.

Lemma 7.12. *Given any weakly connected graph $G = (V, E_L \cup E_Q \cup E_{DB})$. Eventually, the explicit edges in E_L form a sorted list.*

Proof. First note that we only downgrade edges from E_Q and E_{DB} to E_L via Linearize, but never *upgrade* edges from E_L to either E_Q or E_{DB}; i.e., BuildList does not call any of the actions from the q-neighborhood sub-protocol or the De Bruijn sub-protocol. Now observe that for any edge $(u, v) \in E_Q$, u is eventually downgraded to BuildList, namely at the time where it is chosen in u's Timeout action in Algorithm 12. Similarly, observe that for any edge $(u, v) \in E_{DB}$, u is eventually downgraded to BuildList, namely at the time where it is chosen in u's Timeout action in Algorithm 14. We can therefore conclude that eventually the subgraph $G' = (V, E_L)$ of G is weakly connected. By Theorem 6.12 the explicit edges in E_L eventually form a sorted list. This finishes the proof. □

We now show that for a fixed value $u.q_r$, $u.Q_r$ eventually contains all nodes in $[u, 2u.q_r]$. This implies that eventually the explicit edges in E_Q form a q-connected list.

Lemma 7.13. *Given any weakly connected graph $G = (V, E_L \cup E_Q \cup E_{DB})$, where the explicit edges in E_L form a sorted list. For a node $u \in V$ with fixed $u.q_r$, the set $u.Q_r$ eventually contains all nodes in $[u, 2u.q_r]$.*

Proof. Assume that for a node $u \in V$ there exists a node $v \in V$ with $v \in [u, 2u.q_r]$ but $v \notin u.Q_r$. W.l.o.g. assume that v is minimal: i.e., out of all nodes w that are contained in $[u, 2u.q_r]$ but not in $u.Q_r$ v minimizes $|u - w|$. As v is minimal and the sorted list already has converged, it holds that $v = w.right$ for a node $w \in u.Q_r \cup \{u\}$. Assume that $w \neq u$, as in this case u already knows v and thus includes it into $u.Q_r$ in the Timeout action of Algorithm 12. Eventually w is picked by u in u's Timeout action and thus u introduces w's left list neighbor w' to w via the Introduce(w', u) action of Algorithm 12. Upon processing the action Introduce(w', u), w sends back its right list neighbor v to u by calling the action Introduce(v, $\perp$) on u. This generates an implicit edge from u to v, which is then transformed into an explicit edge because u then includes v into $u.Q_r$ upon processing the action Introduce(v, $\perp$). □

One can easily show an analogous statement for the set $u.Q_l$.

Now we show that eventually $u.q_r$ becomes stable. The proof for $u.q_l$ works analogously. In order to show that eventually $u.q_r$ becomes stable, we need the following technical lemma.

Lemma 7.14. *Let $\zeta = 8c$ for some constant $c \geq 1$ and $x \geq \zeta \log n$ and $y \in [0, 1)$. Let $I = [y, y + \frac{x}{n}]$ be an interval over $[0, 1)$ of size $\frac{x}{n}$. Then the number of nodes with a label in I is within $\left(\frac{1}{2}x, 2x\right) \in \Theta(x)$ w.h.p.*

Proof. W.l.o.g. assume that $x = \zeta \log n$. For all $u \in V$ let X_u be a binary random variable with

$$X_u = \begin{cases} 1, & \text{if } u \in I \\ 0, & \text{otherwise.} \end{cases}$$

Then it holds that $\Pr[X_u = 1] = \frac{x}{n}$ and for $X := \sum_{u \in V} X_u$ it holds that $\mu = \mathbb{E}[X] = n \cdot \frac{x}{n} = x$. $\mathbb{E}[X]$ represents the expected number of nodes with a label in I. Now fix $\delta = 1$. By following the Chernoff bound from Theorem 2.14(a) we get that

$$\begin{aligned}\Pr[X \geq (1+\delta)\cdot\mu] &\leq \exp\left(\frac{-1^2 \cdot x}{3}\right)\\ &= \exp\left(\frac{-\zeta \log n}{3}\right)\\ &< \exp\left(-c \log n\right)\\ &\leq n^{-c}.\end{aligned}$$

Analogously, by following the Chernoff bound from Theorem 2.14(b) we get for $\delta = \frac{1}{2}$ that

$$\begin{aligned}\Pr[X \leq (1-\delta)\cdot\mu] &\leq \exp\left(\frac{-\left(\frac{1}{2}\right)^2 \cdot x}{2}\right)\\ &= \exp\left(\frac{-\zeta \log n}{8}\right)\\ &= \exp\left(-c \log n\right)\\ &\leq n^{-c}.\end{aligned}$$

This proofs the lemma. □

We are now ready to show that $u.q_r$ eventually yields the desired value of $\Theta(\sqrt[d]{n})$ and is not updated afterwards anymore: i.e., it becomes stable.

Lemma 7.15. *Consider a sorted list over the interval $[0,1)$ and a node $u \in V$. Eventually $u.q_r \in \left(\frac{\sqrt[d]{n}}{8n}, \frac{\sqrt[d]{n}}{2n}\right) = \Theta(\sqrt[d]{n})$ w.h.p. and $u.q$ does not get updated anymore as long as no nodes join or leave the system.*

Proof. Recall the function $f_r(x)$, the interval $I_r(x)$ and the set $N_r(x)$ from Algorithm 13. Lemma 7.14 implies that $N_r(x) > N_r(x+1)$ if $|I_r(x)| = \frac{1}{2^x} \in \Omega\left(\frac{\log n}{n}\right)$. We show that $f_r(x)$ is monotonically decreasing when considering values for x such that $\frac{1}{2^x} \geq \frac{\log n}{n}$. For this we show that $f_r(x) > f_r(x+1)$. We get

$$\begin{aligned}& f_r(x) > f_r(x+1)\\ \Leftrightarrow\quad & \left(\frac{4}{\zeta}\right)^d \cdot |I_r(x)| - \left(\frac{1}{N_r(x)}\right)^{d-1} > \left(\frac{4}{\zeta}\right)^d \cdot |I_r(x+1)| - \left(\frac{1}{N_r(x+1)}\right)^{d-1}\\ \Leftrightarrow\quad & \left(\frac{4}{\zeta}\right)^d \cdot \frac{1}{2^x} - \left(\frac{1}{N_r(x)}\right)^{d-1} > \left(\frac{4}{\zeta}\right)^d \cdot \frac{1}{2^{x+1}} - \left(\frac{1}{N_r(x+1)}\right)^{d-1}\\ \Leftrightarrow\quad & \left(\frac{1}{N_r(x+1)}\right)^{d-1} - \left(\frac{1}{N_r(x)}\right)^{d-1} > \left(\frac{4}{\zeta}\right)^d \cdot \frac{1}{2^{x+1}} - \left(\frac{4}{\zeta}\right)^d \cdot \frac{1}{2^x}\\ \Leftrightarrow\quad & \underbrace{\left(\frac{1}{N_r(x+1)}\right)^{d-1} - \left(\frac{1}{N_r(x)}\right)^{d-1}}_{\geq 0} > \underbrace{-\left(\frac{4}{\zeta}\right)^d \cdot \frac{1}{2^{x+1}}}_{< 0}.\end{aligned}$$

Now note that $f_r(x) = 0$ if $\frac{1}{2^x} = \frac{\zeta \sqrt[d]{n}}{4n}$ because of

$$\begin{aligned}
\left(\frac{4}{\zeta}\right)^d \cdot \frac{1}{2^x} - \left(\frac{1}{N_r(x)}\right)^{d-1} &\overset{Lemma\ 7.14}{=} \left(\frac{4}{\zeta}\right)^d \cdot \frac{\zeta \sqrt[d]{n}}{4n} - \left(\frac{4}{\zeta \sqrt[d]{n}}\right)^{d-1} \\
&= \left(\frac{4}{\zeta}\right)^{d-1} \frac{\sqrt[d]{n}}{n} - \frac{4^{d-1}}{\zeta^{d-1} \sqrt[d]{n}^{d-1}} \\
&= \left(\frac{4}{\zeta}\right)^{d-1} \frac{\sqrt[d]{n}}{n} - \left(\frac{4}{\zeta}\right)^{d-1} \frac{\sqrt[d]{n}}{n} \\
&= 0.
\end{aligned}$$

Since each node u keeps all its neighbors that are within the interval $[u, u + 2u.q_r] = [u, u + \frac{1}{2^{x-1}}]$ it follows that u is able to compute at least the functions $f_r(x)$, $f_r(x-1)$ and $f_r(x+1)$ at any point in time. This way u can decide whether to increase, decrease or keep the current value of x and thus whether or not the value $u.q_r$ has to be updated. Therefore, $u.q_r$ eventually will contain a value $\frac{1}{2^x}$ for which it holds that $\frac{\sqrt[d]{n}}{8n} < \frac{1}{2^x} < \frac{\sqrt[d]{n}}{2n}$, which proves the lemma. Note that the above computations do not hold in cases where $u.q_r = \frac{1}{2^x} < \frac{\log n}{n}$. However, since we make sure in our algorithm that we always compute x such that $x < N_r(x)/c - \log N_r(x)$ holds, it follows by Lemma 7.10 that $u.q_r = \frac{1}{2^x} \geq \frac{\log n}{n}$ at any point in time. □

Finally, we show that once the sorted list and the q-connected list have converged, all correct De Bruijn edges are eventually generated.

Lemma 7.16. *Given any weakly connected graph $G = (V, E_L \cup E_Q \cup E_{DB})$, where the explicit edges in E_L form a sorted list and the explicit edges in E_Q form a $\Theta(\sqrt[d]{n})$-connected list. Eventually the De Bruijn edges E_{DB} are set up correctly according to Definition* 7.9(d).

Proof. Assume a node $u \in V$ generates a probe $M(i, j)$ in the Timeout action of Algorithm 14. Then $M(i, j)$ is first forwarded to the node $u.db(i-1, k), k = j \bmod 2^{i-1}$, or via a list edge and then forwarded via one standard De Bruijn hop. In case the first and/or the second step is not possible (other standard De Bruijn edges may not have been set up yet), the algorithm proceeds with a greedy search for the node v that is closest to the point $\frac{u+j}{2^i}$. Since the explicit edges in E_Q already form a $\Theta(\sqrt[d]{n})$-connected list, the greedy search is successful, so the correct node v is sent back to u, which leads to u correctly updating $u.db(1, j)$.

As u eventually has generated a probe $M(i, j)$ for every pair (i, j) such that $i \in \{1, \ldots, \lceil \log(\sqrt[d]{n}) \rceil\}$ and $j \in \{0, \ldots, 2^i - 1\}$, all De Bruijn edges outgoing at u are eventually established. □

By all of the above lemmas we conclude the following lemma:

Lemma 7.17 (Convergence)**.** *BuildGDB transforms any weakly connected graph $G = (V, E_L \cup E_Q \cup E_{DB})$ into a GDB.*

We are now ready to show the closure property:

Lemma 7.18 (Closure)**.** *If the explicit edges in $G = (V, E_L \cup E_Q \cup E_{DB})$ already form a GDB, then they are preserved at any point in time if no nodes join or leave the system.*

Proof. For the list edges E_L closure follows from Theorem 6.12 because the BuildList protocol modified edges only if a node u gets to know a node v with either $u.left \prec v \prec u$ or $u \prec v \prec u.right$, which is not possible, because $u.left$ and $u.right$ already store u's closest list neighbors.

For the edges in E_Q we know that none of these edges are removed or forwarded by any node u since u's variables $u.q_l, u.q_r \in \Theta(\sqrt[d]{n})$ do not change as long as no nodes join or leave the system (Lemma 7.15).

Aside from the initial checks in the Timeout action of Algorithm 14, every node u modifies its variables $u.db(i,j)$ only via the action ProbeDone of Algorithm 14 under the condition that the probe result is different from the node currently stored in $u.db(i,j)$. Since the system is already in a legitimate state, all probes $M(i,j)$ return the node that is closest to the target position of $M(i,j)$, so the result of a probe does not change: $u.db(i,j)$ already stores the node that is closest to the point $\frac{u+j}{2^i}$. Also, no variable $u.db(i,j)$ is updated in the first command of the Timeout action of Algorithm 14, as u's approximation of $\Theta(\sqrt[d]{n})$ does not change. This concludes the proof. □

Lemmas 7.17 and 7.18 together imply our main result of this section:

Theorem 7.19. *BuildGDB is self-stabilizing.*

Additional Properties

In this section we show some further properties for our system that hold in legitimate states.

The following theorem shows that the probing approach for De Bruijn edges is efficient in legitimate states regarding the number of hops a single probe $M(i,j)$ has to perform:

Theorem 7.20. *Let the GDB G be in a legitimate state. A probe $M(i,j)$ generated at node u only needs 3 hops w.h.p. to be routed u to the node v that is closest to the point $\frac{u+j}{2^i}$.*

Proof. We show the lemma for $i > 1$ and $j < 2^{i-1}$. The proofs for the other cases work analogously. Node $u \in V$ forwards the probe $M(i,j)$ to the node $v_1 = u.db(i-1,j)$, i.e., to the node v_1 that is closest to the point $\frac{u+j}{2^{i-1}}$. At v_1 we then perform a standard De Bruijn hop by forwarding $M(i,j)$ to the node $v_2 = v_1.db(1,0)$. If nodes are distributed perfectly on the $[0,1)$-interval it would hold $v = v_2$ with

$$h(v_2) = \frac{h(v_1) + j}{2^i} = \frac{\frac{u+j}{2^{i-1}} + 0}{2} = \frac{u+j}{2^i}.$$

However, by Lemma 7.7, $h(v_1)$ may deviate from the value $\frac{u+j}{2^{i-1}}$ by at most $\Theta\left(\frac{\log n}{n}\right)$ w.h.p. Assume w.l.o.g. that $h(v_1) = \frac{u+j}{2^{i-1}} + \frac{\log n}{n}$. When forwarding $M(i,j)$ from v_1

to v_2 we may again deviate by at most $\frac{\log n}{n}$. Consequently, w.l.o.g., $h(v_2)$ is equal to

$$\frac{h(v_1)+j}{2^i}+\frac{\log n}{n}=\frac{\frac{u+j}{2^{i-1}}+\frac{\log n}{n}+j}{2^i}+\frac{\log n}{n}=\frac{u+j}{2^i}+\frac{3\log n}{2n}.$$

v_2 maintains a q-neighborhood; i.e., it stores all nodes within the interval $[v_2 - v_2.q_l, v_2 + v_2.q_r]$. Since it holds that $v_2.q \in \Theta(\sqrt[d]{n})$ and $\frac{3\log n}{2n} \in \mathcal{O}(\sqrt[d]{n})$, the node v closest to the point $\frac{u+j}{2^i}$ is contained in v's q-neighborhood, so it is reached via one hop performed in the third step of the probing. □

Next we show that the outdegree of the GDB matches the degree of the generalized $\sqrt[d]{n}$-ary, d-dimensional De Bruijn graph (Definition 7.1) asymptotically:

Theorem 7.21. *Each node in the GDB has outdegree $\Theta(\sqrt[d]{n})$ w.h.p.*

Proof. We count the number of edges for a node $u \in V$ according to Definition 7.4. u has edges to all nodes in the interval $[u - 2u.q_l, u + 2u.q_r]$, which are $\Theta(\sqrt[d]{n})$ nodes w.h.p. according to Lemmas 7.14 and 7.15. This covers all outgoing edges of u in $E_L \cup E_q$.

Next, count the number of De Bruijn edges outgoing at u for each level $i = 1, \ldots, \log q$: On level i, there are exactly 2^i De Bruijn edges, leading from q generalized De Bruijn edges on level $\log(q)$ to the 2 standard De Bruijn edges on level 1. Therefore, u has

$$\begin{aligned}\sum_{i=1}^{\log(q)} 2^i &= \left(\sum_{i=0}^{\log(q)} 2^i\right) - 1\\ &= \left(\frac{1-2^{\log(q)+1}}{1-2}\right) - 1\\ &= (2q-1)-1\\ &= 2q-2\end{aligned}$$

De Bruijn edges.

Summing everything up results in u having $\Theta(\sqrt[d]{n})$ outgoing edges. □

Finally, we want to investigate the performance of BuildGDB in the case where new nodes join the system. For this we assume that a node v may join the system by introducing itself to an arbitrary (old) node u. We distinguish between the sets V_{new} and V_{old}, where V_{new} consists of all nodes that joined the system and V_{old} consists of all nodes that are already part of the system. The nodes in V_{old} already form a legitimate GDB at the time when nodes in V_{new} join. Define the *work* $W(u)$ for a node $u \in V_{old}$ as the number of explicit edges that it needs to build or redirect when new nodes join the system.

We first introduce the following two technical lemmas:

Lemma 7.22. *The expected distance between two neighbors $u, v \in V$ on the $[0,1)$-interval via a pseudorandom hash function $h : \mathbb{N} \to [0,1)$ is equal to $\frac{1}{n}$.*

Proof. Let $X_1, \ldots, X_n$ be the hashes of n nodes when using h. W.l.o.g. assume that there exists $i \in \{1, \ldots, n\}$ with $X_i = 0$. The cumulative distribution function (CDF) $F_{X_j}(x)$ is defined by

$$F_{X_j}(x) = \Pr[X_j \leq x]$$

for an arbitrary $j \in \{1, \ldots, n\}$. It holds that the CDF of $X_{\min} = \min\{\{X_1, \ldots, X_n\} \setminus \{X_i\}\}$ is equal to the probability that not all X_j's are greater than X; i.e., it is given by

$$F_{X_{\min}}(x) = 1 - (1 - \Pr[X_{\min} \leq x])^{n-1}.$$

Since

$$\Pr[X_j \leq x] = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x \leq 1 \\ 1, & x > 1 \end{cases}$$

it follows

$$F_{X_{\min}}(x) = \begin{cases} 0, & x < 0 \\ 1 - (1-x)^{n-1}, & 0 \leq x \leq 1 \\ 1, & x > 1 \end{cases}$$

We need to compute the expected distance between $X_i = 0$ and $X_{\min}$, since these hashes belong to neighboring nodes. It holds that

$$\begin{aligned} \mathbb{E}[|X_{\min} - X_i|] = \mathbb{E}[X_{\min}] &= \int_0^\infty 1 - F_{X_{\min}}(t)\ dt \\ &= \int_0^1 1 - (1 - (1-t)^{n-1})\ dt \\ &= \int_0^1 (1-t)^{n-1}\ dt = \left[-\frac{1}{n}(1-t)^n\right]_0^1 = \frac{1}{n}. \end{aligned}$$

To see that $\mathbb{E}[X] = \int_0^\infty 1 - F_X(t)\ dt$ for a non-negative random variable X consider, for example, the book of Ghahramani [Gha05]. □

Lemma 7.23. *Consider n nodes that have been hashed via a pseudorandom hash function $h : \mathbb{N} \to [0, 1)$. Let $p \in [0, 1)$ be an arbitrary point. The expected distance between p and the node that is located closest to p is no larger than $\frac{1}{2n}$.*

Proof. Let the point p lie between two consecutive nodes v_1 and v_2. From Lemma 7.22 we know that the expected distance between v_1 and v_2 is $\frac{1}{n}$. The distance between p and the node that is closest to p is maximized when p lies exactly in the middle between v_1 and v_2: i.e., $p = \frac{v_1+v_2}{2}$. This distance is equal to $\frac{1}{2n}$. □

Theorem 7.24. *Let the GDB G be in a legitimate state. When n increases by factor 2^d, i.e., $|V_{old}| = n$ and $|V_{new}| = 2^d \cdot n - n$, then for a node $u \in V_{old}$ it holds w.h.p. that $W(u) \in \Theta(\sqrt[d]{n})$, which is asymptotically optimal.*

Proof. Having n being increased by factor 2^d results in $\sqrt[d]{n}$ getting increased by factor 2. Each node $u \in V_{old}$ therefore increases its approximation of $\sqrt[d]{n}$ by factor 2, meaning that u also increases the size of its interval $I = [u - u.q_l, u + u.q_r]$ by factor 2. Due to Lemma 7.14 this results in u having to build $\Theta(\sqrt[d]{n})$ new edges for its

q-neighborhood. Also, since $u.q$ increases by factor 2, u builds a new level of De Bruijn edges: i.e., u generates variables $u.db(\log(u.q)+1, j)$ for all $j \in \{0, \ldots, 2\log(u.q)-1\}$, which are $2\log(u.q) - 1 = \Theta\left(\frac{1}{d}\log n\right) = \Theta(\log n)$ many.

Now compute the number of De Bruijn edges that need to be redirected by u. Consider an arbitrary De Bruijn edge $\frac{u+j}{2^i}$ with $i \in \{1, \ldots, \log(u.q)\}$ and $j \in \{0, \ldots 2^i - 1\}$. The associated node to this edge is $u.db(i, j)$. W.l.o.g. let $u.db(i, j) \geq \frac{u+j}{2^i}$. In order for the edge $(u, u.db(i, j))$ to be redirected from $u.db(i, j)$ to a new node $v \in V_{new}$, v has to join within the interval $\left[\frac{u+j}{2^i} - \left(u.db(i, j) - \frac{u+j}{2^i}\right), u.db(i, j)\right]$. From Lemma 7.23 we know that the expected size of this interval is no larger than $2 \cdot \frac{1}{2n} = \frac{1}{n}$. Denote by the set I all of the intervals mentioned above and assume the size of an interval $i \in I$ is of the form $\frac{X_i}{n}$ for random variable $X_i \in \left[\frac{1}{n}, \log n\right]$ with $\mathbb{E}[X_i] = 1$. Remember that we have to consider $\Theta(\sqrt[d]{n})$ of those intervals (one for each of u's De Bruijn edges), so $|I| = \Theta(\sqrt[d]{n})$. Let $X = \sum_{i \in I} X_i$. Then $\mathbb{E}[X] = \Theta(\sqrt[d]{n})$. W.l.o.g. let $\mathbb{E}[X] = \sqrt[d]{n}$. Choose $\delta = \frac{\sqrt{3c \cdot \log n}}{d(\sqrt[2]{n})}$ for some constant c. For n high enough, it holds that $\delta \leq 1$. Following Theorem 2.14(a), we get

$$\begin{aligned}
\Pr[X \geq (1+\delta) \cdot \mathbb{E}[X]] &\leq \exp\left(\frac{-\left(\frac{\sqrt{3c \cdot \log n}}{d(\sqrt[2]{n})}\right)^2 \cdot \sqrt[d]{n}}{3}\right) \\
&= \exp\left(-c \log n\right) \\
&\leq n^{-c}.
\end{aligned}$$

Thus, w.h.p., it holds that X is upper bounded by $\mathcal{O}(\sqrt[d]{n})$ and therefore, the sum of the sizes of the intervals in I is upper bounded by $\mathcal{O}\left(\frac{\sqrt[d]{n}}{n}\right)$.

We show that now u has to redirect up to $\mathcal{O}(\sqrt[d]{n})$ edges w.h.p. For each joining node v define Y_v as

$$Y_v = \begin{cases} 1, & \text{if } v \in V_{new} \text{ joins into one of the intvervals in } I \\ 0, & \text{otherwise.} \end{cases}$$

Assume w.l.o.g. that $\Pr[Y_v = 1] = \frac{\sqrt[d]{n}}{n}$. Then $Y = \sum_{v \in V_{new}} Y_v$ and thus,

$$\mathbb{E}[Y] = (2^d \cdot n - n) \cdot \frac{\sqrt[d]{n}}{n} = (2^d - 1) \cdot \sqrt[d]{n}.$$

Next, choose $\delta = \frac{\sqrt{3c \cdot \log n}}{\sqrt{2^d - 1} \cdot d(\sqrt[2]{n})}$. Following Theorem 2.14(a), we get

$$\begin{aligned}
\Pr[Y \geq (1+\delta) \cdot \mathbb{E}[Y]] &\leq \exp\left(\frac{-\left(\frac{\sqrt{3c \cdot \log n}}{\sqrt{2^d - 1} \cdot d(\sqrt[2]{n})}\right)^2 \cdot (2^d - 1) \cdot \sqrt[d]{n}}{3}\right) \\
&= \exp\left(-c \log n\right) \\
&\leq n^{-c}.
\end{aligned}$$

Therefore, the number of nodes $v \in V_{new}$ that join within intervals in I, and thus have to be redirected by u, is upper bounded by $(1+\delta)\mathbb{E}[Y] \leq 2\mathbb{E}[Y] = \mathcal{O}(\sqrt[d]{n})$ w.h.p. for n high enough.

Putting everything together, we get that $W(u) \in \Theta(\sqrt[d]{n})$ w.h.p.

At last, we argue why the work of $\Theta(\sqrt[d]{n})$ is asymptotically optimal. If n increases by factor 2^d, the systems contains $2^d \cdot n$ nodes. Since the degree is still d, Fact 7.2 implies that there has to be a node of degree $2\sqrt[d]{n}$, which leads to at least one old node $u \in V_{old}$ having to double its degree of $\sqrt[d]{n}$. Therefore, v has to create at least $\sqrt[d]{n}$ new edges and thus spend $\Omega(\sqrt[d]{n})$ amount of work. □

CHAPTER 8

Self-Stabilizing Quadtrees

Most protocols in topological self-stabilization (including the one from the previous chapter) only show that the system *eventually* converges to a legitimate state, without considering the *monotonicity* of the actual recovery process. Monotonicity means that the functionality of the system regarding a specific property never gets worse as time progresses; i.e., for two points in time t, t' with $t < t'$, the functionality of the system w.r.t. a specific property is better in t' than in t.

In this chapter we are interested in *searching*, as this is one of the most important operations in a distributed system. We study systems that satisfy *monotonic searchability*: If a search request for node v starting at node u succeeds at time t, then every search request for v initiated by u at time $t' > t$ succeeds as well.

Previous work on monotonic searchability [SSS15; SSS16] proposes self-stabilizing protocols for one-dimensional topologies (for instance, a sorted list). Still, up to this point it is not known how to come up with an efficient self-stabilizing protocol for high-dimensional settings that satisfies monotonic searchability. High-dimensional settings are relevant in areas such as wireless ad-hoc networks or social networks where processes are defined by multiple parameters.

This paper introduces a novel protocol BuildQT for a self-stabilizing *quadtree* along with a routing protocol SearchQT that satisfies monotonic searchability and terminates after $\mathcal{O}(\log n)$ hops on any input. To the best of our knowledge, this is the first protocol that combines self-stabilization and monotonic searchability for the two-dimensional case. In addition, one can easily extend our protocols in order to work for multiple dimensions, leading to a self-stabilizing *octree*. We furthermore expand the notion of monotonic searchability to an even stronger and more realistic property, which we call *geographic monotonic searchability*, and show that SearchQT satisfies this property as well. Our protocols stand out due to their simplicity and elegance and do not require restrictive assumptions on messages, as it has been done for the universal approach [SSS16].

Underlying Publication. This chapter is based on the following publication:

> M. Feldmann, C. Kolb, and C. 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)*, 2018, cf. [FKS18].

Outline of This Chapter. We first formally define the concept of monotonic searchability and its geographical extension in Section 8.1. Then we provide some additional related work in Section 8.2 and describe the quadtree topology that we want the

nodes to form in legitimate states (Section 8.3). Afterwards, we present the protocol BuildQT (Section 8.4) along with the routing protocol SearchQT (Section 8.5). We show that BuildQT is self-stabilizing and, when combined with SearchQT, satisfies geographic monotonic searchability. Finally, we show how to extend BuildQT in order to work for arbitrarily high dimensions, leading to self-stabilizing octrees (Section 8.6).

8.1. Monotonic Searchability

Before we can define monotonic searchability and its geographical extension, we need to introduce some notation and assumptions. Until Section 8.6 we consider a two-dimensional square P of unit side length. Each node $u \in V$ is represented by its *unique position* in P given by *coordinates* $(u_x, u_y) \in [0,1]^2$. Like the identifier of a node, coordinates are assumed to be read-only; i.e., there are no coordinates of non-existing nodes in the initial state of the system. Having node coordinates be read-only also makes sense in our setting, as these are usually delivered by an external component that is not in control of our protocol, such as GPS, for instance. Define $||uv||_2$ as the Euclidean distance between two nodes $u, v \in V$, i.e., $||uv||_2 = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}$.

Nodes are able to issue search requests at any point in time:

Definition 8.1. *A* search request *is a message* $\mathsf{Search}(u, (x, y))$*, where* u *is the sender of the message and* $(x, y) \in [0,1]^2$ *are the coordinates* u *wants to search for.*

A search request is delegated along edges in G according to a given routing protocol, until the request *terminates*: i.e., either the node with coordinates (x, y) is reached or the request cannot be forwarded anymore. Note that (x, y) do not necessarily need to be coordinates of an existing node. In such a case, the routing protocol may just stop at some node v that would have been on the routing path to node u if node u with coordinates (x, y) had existed. Upon termination at node v, the reference of v is returned to the sender u (in the pseudocode we indicate this via a return statement).

We consider the following definition of monotonic searchability:

Definition 8.2 (Monotonic Searchability)**.** *A self-stabilizing protocol satisfies* monotonic searchability *according to some routing protocol* $\mathcal{R}$*, if it holds for any pair of nodes* $u, v \in V$ *that once a search request* $\mathsf{Search}(u, (v_x, v_y))$ *returns* v *at time* t*, any search request* $\mathsf{Search}(u, (v_x, v_y))$ *initiated at time* $t' > t$ *also returns* v*.*

Realizing monotonic searchability in self-stabilizing systems is a non-trivial problem, because once a $\mathsf{Search}(u, (v_x, v_y))$ request returns v to u, it cannot trivially be guaranteed that v is found again by u at later stages, due to the modification of edges by the self-stabilizing protocol.

Definition 8.2 differs in a minor detail compared to the definition stated in [SSS15; SSS16]. The initial search request issued by u terminates at time t, but Scheideler et al. define the time step t to be the one at which the initial search request was generated by u. They use a probing approach to check for a node u whether u is still waiting for the result of a previously issued search request and to cache all search

requests searching for the same target. The same approach can be applied to our protocol as well to overcome this, but for simplicity we use the slightly modified definition stated above.

In two-dimensional scenarios it is more realistic to search for geographic positions rather than for concrete node addresses. To handle this, we introduce the following definition of geographic monotonic searchability.

Definition 8.3 (Geographic Monotonic Searchability)**.** *Let $(x, y) \in [0, 1]^2$ be an arbitrary position in P. Let $v \in V$ be the node that is returned by $\mathsf{Search}(u, (x, y))$ if the system is in a legitimate state. A self-stabilizing protocol satisfies* geographic monotonic searchability *according to some routing protocol $\mathcal{R}$, if in case the system is in an arbitrary state and $\mathsf{Search}(u, (x, y))$ returns v at time t, then any request $\mathsf{Search}(u, (x, y))$ initiated at time $t' > t$ also returns v.*

A protocol satisfying geographic monotonic searchability also satisfies monotonic searchability.

We aim to solve the following problem: Given a weakly connected graph $G = (V, E)$ of n nodes with coordinates in P, construct a self-stabilizing protocol along with a routing protocol such that geographic monotonic searchability is satisfied. The self-stabilizing protocol should transform G into a network in which any search request terminates after $\mathcal{O}(\log n)$ hops, given that the Euclidean distance between any two nodes is at least $1/n$.

8.2. Related Work

This section presents an overview of related work, specifically for quad- and octrees and the concept of monotonic searchability.

Quad- and Octrees. Quadtrees have first been introduced by Finkel and Bentley [FB74]. Since then, quadtrees and octrees are widely used in computational geometry (for surveys consider [Alu04; Sam89], for example). There are peer-to-peer approaches relying on quadtrees [Gao+04; THS07] as well. Still, the problem of designing a self-stabilizing protocol that arranges peers in a quadtree has not been tackled so far.

Monotonic Searchability. Research on monotonic searchability was initiated by Scheideler, Setzer and Strothmann in [SSS15], where the authors present a self-stabilizing protocol for the sorted list that satisfies monotonic searchability. They also showed that providing monotonic searchability is impossible in general when the system contains corrupted messages. However, this property is restricted to cases where the desired topology to which the graph should converge is clearly defined, forcing the underlying protocol to eventually remove an explicit edge if it is not part of the desired topology. This is not the case for our topology, because once a specific explicit edge (which we define as *quad edge* later on) is generated by our protocol it is never deleted, so the legitimate state s that we reach is dependent on the specific computation done before reaching s. We therefore do not need to enforce any restrictions on messages, as routing is done via quad edges only.

Building on their research, the same authors present a universal approach for maintaining monotonic searchability along with a generic routing protocol that can be applied to a wide range of topologies [SSS16]. However, adapting their protocol to specific topologies comes at the cost of convergence times and additional message overhead. This is due to the fact that whenever an explicit edge is delegated from node u to v, u has to wait for an acknowledgment from v until it is allowed to remove the explicit edge from its local storage. Furthermore, a search request that is forwarded via the generic routing protocol might travel $\Omega(n)$ hops when searching for non-existing nodes, whereas our routing protocol only needs $\mathcal{O}(\log n)$ hops on any input to terminate (if the nodes are spread uniformly in the plane), while still satisfying monotonic searchability. In addition to this, our protocol BuildQT is simpler and also more lightweight regarding the message overhead. This is mostly due to the simplicity of the quadtree topology.

A self-stabilizing protocol for the skip graph satisfying monotonic searchability has been presented by Luo et al. [LSS19]. The idea is similar to ours. That is, once an explicit edge has been established it is never deleted in any further computation. In legitimate states, the skip graph is then a subgraph of the final topology, keeping routing paths the same and thus satisfying monotonic searchability. The authors also show how to dismantle additional edges without violating monotonic searchability such that the network converges to a perfect skip graph.

Close but different from our notion of monotonic searchability is the notion of *monotonic stabilization* [YT10]. A self-stabilizing protocol is monotonically stabilizing, if every change done by its nodes is making the system approach a legitimate state and if every node changes its output only once. The authors show that nodes have to exchange additional information in order to satisfy monotonic stabilization.

Interestingly, topological self-stabilization (and monotonic searchability) in two- or high-dimensional settings has barely been investigated until now. There exists a single self-stabilizing protocol that transforms any weakly connected graph into a two-dimensional topology – the *Delaunay Graph* [Jac+12]. Unfortunately, it seems non-trivial to extend this such that monotonic searchability is satisfied, without resorting to expensive mechanisms like broadcasting or the universal protocol from [SSS16].

8.3. Quadtrees

In this section we introduce the quadtree as our desired topology and define legitimate states of our system. We first need some notation: Denote by $P' \subseteq P$ that P' is a *subarea* of P and denote the area covered by two subareas $P_1, P_2 \subseteq$ by $P_1 \cup P_2$. If the coordinates (u_x, u_y) of a node $u \in V$ lie in a (sub-)area $P' \subseteq P$, we say that $u \in P'$. If a subarea $P' \subseteq P$ does not contain a node, we say that P' is *empty*.

Intuitively, our approach works as follows. Given a set V of n nodes with coordinates in P, we first cut the area P into two equally sized *subareas*, by a vertical cut. The resulting subareas $P_1, P_2 \subseteq P$ are then again cut into two equally sized subareas, this time by a horizontal cut. We apply this cutting recursively for each subarea, always alternating between vertical and horizontal cuts. The recursive halving of a subarea is stopped once this subarea contains at most one node. Once cutting is done, we define a total order on all nodes in P similar to following the nodes in P in

a space-filling curve. The total order is then used to connect the nodes into a sorted list via the BuildList protocol (recall Section 6.4). On the basis of this sorted list and the generated subareas, we establish additional edges, which we use for the routing protocol.

More formally, let us first consider the recursive algorithm QuadDivision (see Algorithm 15 for the pseudocode) with parameters $\bar{V} \subseteq V$, $\bar{P} \subseteq P$ and $f \in \{0, 1\}$.

Algorithm 15 Quad Division Algorithm

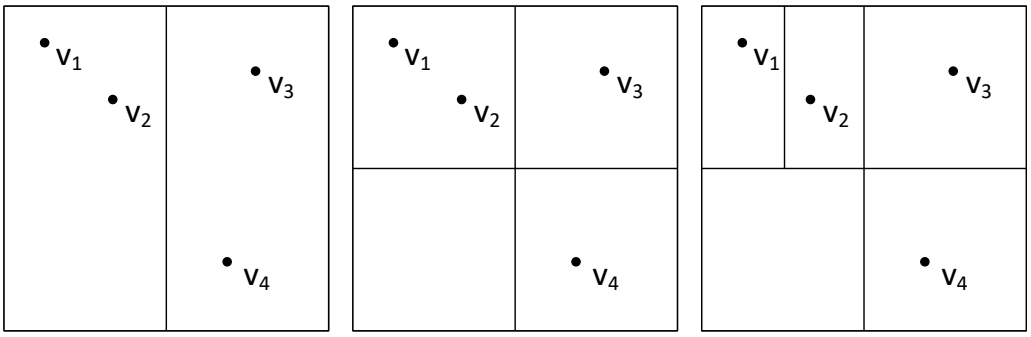
```
1:  QuadDivision(V̄, P̄, f)
2:      if f = 1 then
3:          Perform vertical cut on P̄, resulting in P̄ = P_1 ∪ P_2
4:      else
5:          Perform horizontal cut on P̄, resulting in P̄ = P_1 ∪ P_2
6:      S ← ∅
7:      if |{u ∈ V̄ | u ∈ P_1}| ≤ 1 then
8:          S ← S ∪ {P_1}
9:      else
10:         S ← S ∪ QuadDivision({u ∈ V̄ ∩ P_1}, P_1, ¬f)
11:     if |{u ∈ V̄ | u ∈ P_2}| ≤ 1 then
12:         S ← S ∪ {P_2}
13:     else
14:         S ← S ∪ QuadDivision({u ∈ V̄ ∩ P_2}, P_2, ¬f)
15:     return S
```

Initially we call QuadDivision$(V, P, 1)$ and thus perform a vertical cut on P, dividing it into equally sized subareas P_1 and P_2. Then we call QuadDivision recursively on P_1 and P_2 as long as they contain more than one node. For simplicity, we assume that nodes do not lie on the boundaries of subareas, as this would make the presentation of our algorithm unnecessarily complex. The problem can easily be resolved in practice via a tiebreaker. QuadDivision$(V, P, 1)$ returns the set S of subareas such that

- P' contains at most one node and
- the union of all $P' \in S$ equals P, i.e., $\bigcup_{P' \in S} P' = P$.

Example 8.4. *Figure 8.1 shows an example for a sequence of cuts with* 4 *nodes* $v_1, \ldots, v_4$*. Note that upon termination,* QuadDivision *returns* 5 *subareas (one subarea for each node* v_i *and the empty subarea on the bottom left).*

In the following, we want to view the output of QuadDivision as a binary tree T. The root node corresponds to the entire square P. An inner node of T, corresponding to a (sub-)area P', has two child nodes. Cutting P' into two subareas P_1 and P_2, the *left child* represents the subarea that lies west of the other (when performing a vertical cut on P') or north of the other (when performing a horizontal cut on P'). Similarly, the *right child* represents the subarea that lies east of the other (when performing a vertical cut on P') or south of the other (when performing a horizontal cut on P'). The binary tree is the unique minimal such tree having no leaf node $t \in T$ correspond to a subarea of P that contains more than one node $u \in V$. Note



(a) (b) (c)

Figure 8.1.: Illustration of QuadDivision performed on nodes $v_1, \ldots, v_4$. (a) illustrates the first vertical cut on P. (b) illustrates the horizontal cuts done to subareas P_1 and P_2. (c) illustrates the final vertical cut before termination.

that this makes nodes $u \in V$ correspond to leaf nodes in T, but a leaf node $t \in T$ does not necessarily correspond to a node in V, as the subarea represented by t may be empty. For the rest of this chapter, we refer to the tree T defined above as *area tree*. Figure 8.2a shows the area tree T from Example 8.4.

Using the area tree notation, we obtain a total order on V:

Definition 8.5 (Two-Dimensional Ordering)**.** *Let T be the area tree returned by QuadDivision$(V, P, 1)$. The total order $\prec$ is given by the depth-first search (DFS) traversal of T, searching left childs first.*

Note that the ordering $\prec$ resembles a space-filling curve similar to the Morton-Curve [Mor66]. Other curves like the Hilbert-Curve also work in principle. However, using them makes the presentation of our ideas way more difficult.

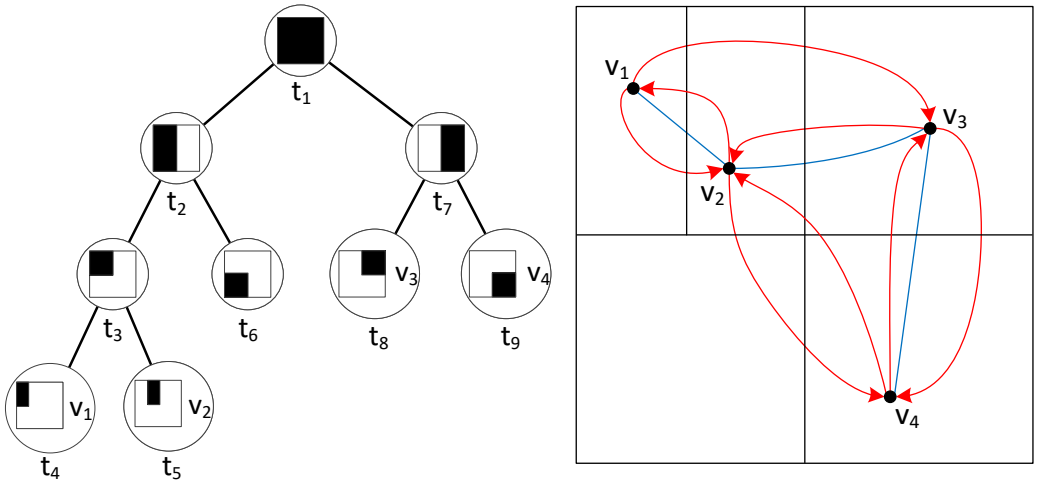
We use the same notation for neighbors in $\prec$ as defined in Definition 6.9.

As nodes in the area tree T correspond to subareas of P and vice versa, we use them interchangeably for the rest of the chapter. We say that a node $t \in T$ *represents* a subarea A, if A is the subarea corresponding to t. The next definition introduces important notations to define the legitimate state of the system:

Definition 8.6. *Let T be the area tree returned by QuadDivision$(V, P, 1)$. For a node $u \in V$, denote the leaf node representing the subarea that contains u by $A(u)$. Consider the unique path $p(u)$ of tree nodes $A(u) = t_1, \ldots, t_k$ from $A(u)$ to the root of T. For each $t_i \in p(u)$ on that path, let s_i be t_i's sibling in T. Define $Q(u) = \bigcup_{i=1}^{k} p_i$.*

It is easy to see that if $t \in Q(u)$, then the subarea represented by t does not contain u, while the subarea represented by the parent node of t contains u. Also, we have that $\left(\bigcup_{A \in Q(u)} A\right) \cup A(u) = P$.

Example 8.7. *Consider again Figure 8.2a: The set $Q(v_1)$ consists of the subareas t_5, t_6 and t_7, as the combination of these with the subarea t_4 containing v_1 yield the square P.*



(a) (b)

Figure 8.2.: (a) Corresponding area tree to Example 8.4. The subareas marked in black are the subareas represented by the corresponding tree node. Performing a depth-first search on the area tree, when always going to the left child first, yields the total order $v_1 \prec v_2 \prec v_3 \prec v_4$. (b) A possible legitimate state for the system from Figure 8.1. List edges are indicated in blue, quad edges in red.

Using the total order $\prec$, we define the legitimate state of our system, i.e., the topology that should be reached by our self-stabilizing protocol BuildQT:

Definition 8.8 (Legitimate State)**.** *The system is in a legitimate state, if the graph induced by the explicit edges satisfies the following conditions:*

(a) *Each node u is connected to its closest left and right neighbor w.r.t. $\prec$.*

(b) *For each non-empty subarea $A \in Q(u)$, u is connected to exactly one node $v \in A$.*

Consider Figure 8.2b showing a possible legitimate state for the nodes from Figure 8.1.

Note that we do not clearly define nodes for u to connect to Definition 8.8(b) more specifically, as we just want to make sure that u is able to reach the subarea directly via an outgoing edge in case the subarea contains nodes. As it will turn out, this helps us to achieve geometric monotonic searchability. We want to emphasize that edges in T are not part of the legitimate state, as we use the area tree to illustrate our approach and only let nodes compute necessary parts of the area tree locally.

8.4. Self-Stabilizing Quadtrees

In this section we describe the self-stabilizing protocol BuildQT and later show that BuildQT is indeed self-stabilizing according to Definition 8.8.

8.4.1. Protocol BuildQT

We first define the protocol-based variables for each node.

Definition 8.9. *For the BuildQT protocol, each node $u \in V$ maintains the following variables:*

(a) *Variables $u.left, u.right \in V \cup \{\perp\}$ storing u's left and right neighbor, respectively.*

(b) *A set $u.Q \subset V$ storing a single node $v \in V$ for each non-empty subarea $A \in Q(u)$ such that $v \in A$.*

We refer to the edges represented by variables $u.left$ and $u.right$ as *list edges* and to edges (u, v) with $v \in u.Q$ as *quad edges*. Observe that an edge (u, v) can be both a list and a quad edge at the same time. The reason for this is that we allow the delegation of search messages only via quad edges (as we will see in Section 8.5), so if u wants to delegate a search message to the subarea containing one of its list edges, it has to make sure that there is a node in $u.Q$ for this area.

Before we can describe how we establish the correct list and quad edges, we shortly describe how a node u that knows some node v is able to locally determine whether $u \prec v$ or $v \prec u$ holds: u just calls QuadDivision($\{u, v\}$, P, 1) locally and obtains an area tree with subareas containing u and v as leaf nodes. Performing a DFS on that tree as described earlier yields either $u \prec v$ or $v \prec u$.

It is important to note that using the same approach, u is also able to compute the set $Q(u)$ for the current system state: u just calls QuadDivision($\{u, u.left, u.right\}$, P, 1). It is easy to see that the corresponding area tree contains all nodes representing subareas in $Q(u)$, so u just has to check each node in the area tree for the properties from Definition 8.6. Obviously, as long as $u.left$ and $u.right$ are still subject to changes, $Q(u)$ also changes. But we will show later that by the way we define our protocol, $Q(u)$ monotonically increases w.r.t. the $\subset$ relation, s.t. none of the proposed properties are violated.

For list edges, we use the BuildList protocol (Section 6.4) using the variables $u.left, u.right$ and the total order $\prec$.

We now describe how we build the correct quad edges at each node. Note that u can easily check whether there exists a subarea $A \in Q(u)$ for which u does not yet have a quad edge, by assigning each $v \in u.Q$ to the subarea in $Q(u)$ that contains v.

The protocol consists of actions Timeout and QLinearize (see Algorithm 16). Before executing any statement of any of these actions, a node u always checks its set $u.Q$ for consistency, ensuring that no two nodes $v_1, v_2 \in u.Q$ are contained in the same subarea $A \in Q(u)$. In case u finds out that $v_1, \ldots, v_k \in u.Q$ are contained in the same subarea $A \in Q(u)$ (which may happen in an initial state), u only keeps one of these nodes (arbitrarily chosen) and forwards all other nodes v_i to BuildList by calling Linearize(v_i).

In Timeout, u chooses a node v from its set $u.Q$ in round-robin fashion and forwards v to BuildList. This has to be done to ensure that the sorted list converges even if the initial weakly connected graph consists of quad edges only. Afterwards, u introduces itself to its left and right neighbors $u.left$ and $u.right$ by calling QLinearize on them.

As part of the same QLinearize request, u asks these nodes if they know a node $v \in A$, where $A \in Q(u)$ is a subarea, for which u does not have a quad edge yet. If this is the case, then u will receive a QLinearize call containing the desired node v as the answer. The subarea A is chosen in round-robin fashion as well, such that each subarea, for which u does not have a quad edge yet, is chosen by u eventually in Timeout. The reason for choosing nodes and subareas in round-robin fashion is that we do not want to overload the network with too many stabilization messages that are generated periodically.

Processing a QLinearize(v, A) request at node u works as follows. We forward v to BuildList and then check if v is contained in a subarea $A' \in Q(u)$ for which there does not exist a node $v' \in u.Q$ with $v' \in A'$. If this is the case, then u does not have a quad edge to the subarea A' yet, so u includes v into $u.Q$, which corresponds to u generating a new quad edge (u, v). Finally, u generates an answer to v as already described above in case u knows a node (including itself) that is contained in A.

Algorithm 16 Protocol BuildQT, executed by node $u \in V$

```
1:  Timeout → true
2:      Consistency check for u.Q
3:      Choose v ∈ u.Q in round-robin fashion and call Linearize(v)
4:      Determine A(u) and Q(u) via QuadDivision({u, u.left, u.right}, P, 1)
5:      Choose A ∈ Q(u) in round-robin fashion s.t. ∀v ∈ u.Q : v ∉ A
6:      u.left ← QLinearize(u, A)                    ▷ A = ⊥ if no such A exists
7:      u.right ← QLinearize(u, A)

8:  QLinearize(v, A)
9:      Consistency check for u.Q
10:     Linearize(v)
11:     Determine A(u) and Q(u) via QuadDivision({u, u.left, u.right}, P, 1)
12:     if ∃A' ∈ Q(u) ∀v' ∈ u.Q : v' ∉ A' then
13:         u.Q ← u.Q ∪ {v}
14:     if A ≠ ⊥ ∧ ∃v' ∈ u.Q ∪ {u} : v' ∈ A then
15:         v ← QLinearize(v', ⊥)
```

8.4.2. Analysis

We show that BuildQT is self-stabilizing according to Definition 8.8.

Recall that our system is initially given by an arbitrary weakly connected graph $G = (V, E)$. As the graph may consist of both list and quad edges, we denote the set of list edges by E_L and the set of quad edges by E_Q, so $G = (V, E_L \cup E_Q)$. Since each node u eventually executes its Timeout action, we assume that no inconsistencies appear, like $u \prec u.left$, $u.right \prec u$ or u having multiple quad edges into the same subarea. We first argue that we get rid of corrupted messages that may exist in an initial state of the system:

Lemma 8.10. *Given any weakly connected graph $G = (V, E_L \cup E_Q)$ and a set of corrupted messages M spread arbitrarily over all node channels. Eventually, G is*

free of corrupted messages, while staying weakly connected.

Proof. By definition of BuildQT we do not delete any node but only forward its node references to BuildList keeping G weakly connected at any point in time. Also notice that a corrupted message $m \in M$ cannot be delegated infinitely by the way we defined the Linearize and QLinearize actions. Because we assume fair message receipt, we know that eventually all messages in M will be processed and thus vanish. □

To show the convergence property, we prove convergence and closure for the sorted list and then show that once the sorted list stabilized, all desired quad edges will eventually be established.

Lemma 8.11. *For a weakly connected graph $G = (V, E_L \cup E_Q)$, BuildQT eventually transforms G such that the explicit edges in E_L form a sorted list w.r.t. $\prec$ (Convergence). If the explicit edges in E_L already form a sorted list w.r.t. $\prec$, then they are preserved at any point in time if no nodes join or leave the system (Closure).*

Proof. In Timeout (Algorithm 16) a node u chooses one of its quad edges $(u, v) \in E_Q$ and forwards it to BuildList, creating an implicit list edge $(u, v) \in E_L$. Since we execute Timeout periodically at each node $u \in V$ and choose quad edges in round-robin fashion, it is guaranteed that eventually each quad edge is forwarded to BuildList. This implies that the graph $G' = (V, E_L)$ eventually becomes weakly connected. We can thus apply Theorem 6.12 to show that the sorted list converges.

Closure for the list edges E_L follows directly from Theorem 6.12. □

Lemma 8.12 (Convergence)**.** *Once the edges in E_L induce a sorted list w.r.t. $\prec$, eventually a legitimate state according to Definition 8.8 is reached.*

Proof. Definition 8.8(a) is already satisfied due to Lemma 8.11, so it remains to show Definition 8.8(b). Recall that u is able to compute $A(u)$ and the set of subareas $Q(u)$ by locally executing QuadDivision($\{u, u.left, u.right\}$, P, 1). As the sorted list has already converged, $Q(u)$ does not change anymore. Let $S \subseteq Q(u)$ be the set of subareas that contain at least one node. We show that $u.Q$ eventually contains one node for each of those subareas: i.e., $\forall A \in S\ \exists v \in u.Q : v \in A$ and v is unique. For this we consider an arbitrary subarea $A \in S$ and assume w.l.o.g. that $u \prec v$ for all $v \in A$. Note that since nodes u choose subareas $A \in Q(u)$ in round-robin fashion, it is guaranteed that u chooses A periodically and asks its list neighbor $u.right$ for a node in A as long as u does not have any quad edge to a node in A. Fix the node $v \in A$ such that v is the outmost left node of A in the ordering $\prec$: i.e., $\forall v' \in A, v' \neq v : v \prec v'$. We show that eventually u will receive an implicit edge $(u, v) \in E_Q$ as part of a QLinearize call and will thus add v to $u.Q$, transforming the implicit edge into an explicit one. Fix $k \in \mathbb{N}_0$ and assume that there are k nodes lying between u and v: i.e., $u \prec v_1 \prec \ldots \prec v_k \prec v$. Observe that any node v_i with $u \prec v_i \prec v$ also needs to have a quad edge to the subarea A, since we defined v to be the outmost left node in A. By definition of our protocol, each node v_i in this chain sends out a QLinearize request to $v_i.right$, demanding for a node lying within the subarea A. Thus, v receives such a request from v_k. As $v \in A$, v answers v_k by sending a QLinearize request containing its own reference back to v_k, such that v_k

establishes an explicit quad edge $(v_k, v) \in E_Q$. Once v_k has established this edge, it answers any incoming QLinearize request coming from v_{k-1} and demanding for a node in A by sending a QLinearize request containing v back to v_{k-1}. Note that as long as v_{k-1} does not yet know v, v_k receives such QLinearize requests periodically from v_{k-1}. The chain continues iteratively until u has received v from v_1, which concludes the proof. □

Now we show the closure property for BuildQT.

Lemma 8.13 (Closure)**.** *If the explicit edges in $G = (V, E_L \cup E_Q)$ already form a quadtree, then they are preserved at any point in time if no nodes join or leave the system.*

Proof. Closure for the list edges E_L follows from Lemma 8.11. By the definition of Algorithm 16 it follows that once a quad edge is established, we do not remove it anymore. It is also easy to see that any reference to a node v that is part of a QLinearize call is just forwarded to BuildList by u and is not included into $u.Q$. This holds because for the subarea $A \in Q(u)$ that contains v there already must exist a node $v' \in u.Q$ with $v' \in A$, since otherwise this would violate condition Definition 8.8(b). □

Combining Lemma 8.12 and Lemma 8.13 yields the main result of this section:

Theorem 8.14. *BuildQT is self-stabilizing.*

8.5. Routing

In this section we state the routing protocol SearchQT (see Algorithm 17 for its pseudocode) and later show that SearchQT in combination with BuildQT satisfies geographic monotonic searchability.

8.5.1. Protocol SearchQT

Before a node u processes a search message, it first performs the same consistency checks on its set $u.Q$ as has been described in Section 8.4. This makes sure that our routing protocol is well-defined. Now assume a node u wants to process a SearchQT(v, (x, y)) message. Consider the subarea $A(u)$ and the set $Q(u)$ of subareas as defined in Definition 8.6. u determines the subarea $A(x, y) \in Q(v) \cup \{A(v)\}$ that contains the position (x, y). If $A(x, y) = A(u)$, then the algorithm terminates and returns u itself to v as the result. Otherwise, u delegates the SearchQT(v, (x, y)) message to the node $w \in u.Q$ with $w \in A(x, y)$. If no edge to a node in $A(x, y)$ exists in $u.Q$, then the algorithm terminates and returns u itself to v as the result. Consider Figure 8.3 for some examples.

8.5.2. Analysis

In this section we show that BuildQT along with the routing protocol SearchQT satisfies geographic monotonic searchability and thus also monotonic searchability.

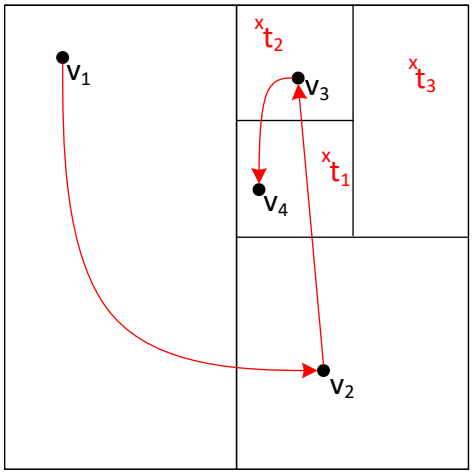


Figure 8.3.: Illustration of the delegation of different SearchQT messages for target coordinates t_1, t_2 and t_3 starting at v_1. SearchQT(v_1,t_1) and SearchQT(v_1,t_2) return the nodes that share the same subarea with the target point (traversing paths (v_1, v_2, v_3, v_4) for t_1 and (v_1, v_2, v_3) for t_2). The search for t_3 yields the path (v_1, v_2, v_3) until SearchQT terminates, as v_3 does not have a quad edge to the subarea containing t_3.

First we need the following technical lemma stating that for each node $u \in V$ the set $Q(u)$ monotonically increases over time:

Lemma 8.15. *Consider an arbitrary system state at time t and a node $u \in V$. Let $Q(u)$ be the output of QuadDivision($\{u, u.left, u.right\}$, P, 1) executed at time t and let $Q(u)'$ be the output of QuadDivision($\{u, u.left, u.right\}$, P, 1) executed at any point in time $t' > t$. Then $Q(u) \subseteq Q(u)'$.*

Proof. By the definition of our protocols, it holds that if u locally calls QuadDivision($\{u, u.left, u.right\}$, P, 1) in order to compute the set $Q(u)$, then any inconsistencies regarding $u.left$ and $u.right$ are already resolved. The lemma then follows from the fact that BuildList does not replace list variables $u.left$ and $u.right$ with nodes that are further away from u than the current entries. More formally, consider w.l.o.g. the variable $u.right$ such that $u \prec u.right$. By the definition of Linearize, u does not replace $u.right$ by a node v for which $u.right \prec v$ holds. This implies that any subsequent QuadDivision($\{u, u.left, u.right\}$, P, 1) call only transfers subareas to $Q(u)$ that are obtained by cutting $A(u)$. Therefore, for any subarea $A \in Q(u)$ we have that $A \in Q(u)'$. □

We are now ready to show the main result of this section:

Theorem 8.16. *BuildQT along with SearchQR satisfies geographic monotonic searchability.*

Algorithm 17 The SearchQT Protocol, executed by node $u \in V$

```
1:  SearchQT(v, (x, y))
2:      Consistency check for u.Q
3:      Determine A(u) and Q(u) via QuadDivision({u, u.left, u.right}, P, 1)
4:      if (x, y) ∈ A(u) then
5:          return u
6:      else
7:          Let A(x, y) ∈ Q(u) with (x, y) ∈ A(x, y)
8:          if ∃w ∈ u.Q : w ∈ A(x, y) then
9:              w ← SearchQT(v, (x, y))
10:         else
11:             return u
```

Proof. Assume a SearchQT(u, (x, y)) request S terminates and returns $v \in V$ to the initiator u at time t, such that v is the node that would have been returned if the system already was in a legitimate state. Now assume that u initiates another SearchQT(u, (x, y)) request S' at time $t' > t$. We show that S' returns v as well.

Let $(u, v_1, \ldots, v_k, v)$ be the path that has been traversed by S. We claim that S' traverses the exact same path as S. Let $Q(u)$ be the output of QuadDivision($\{u, u.left, u.right\}$, P, 1) executed when processing S at u and let $Q(u)'$ be the output of QuadDivision($\{u, u.left, u.right\}$, P, 1) executed when processing S' at u. Let $A(v) \in Q(u)$ be the subarea that contains v and $A(v_1) \in Q(u)$ be the subarea that contains v_1. Since S has been delegated by u to v_1, it follows from the definition of the SearchQuad protocol that $v \in A(v_1)$. Lemma 8.15 implies that $Q(u) \subseteq Q'(u)$ and thus $A(v_1) \in Q'(u)$. It therefore follows from the definition of SearchQT that u delegates S' to v_1 as well. By arguing the same way for any node v_i on the remaining path $(v_1, \ldots, v_k, v)$, we can conclude that S' arrives at v and terminates. This finishes the proof. □

As already indicated in Section 8.1, we obtain the following corollary:

Corollary 8.17. *BuildQT along with SearchQR satisfies monotonic searchability.*

Finally, we show an upper bound on the number of hops for any search message, if we assume that the Euclidean distance $||uv||_2$ between any pair $(u, v) \in V$ is at least $\frac{1}{n}$. We start with the following lemma:

Lemma 8.18. *Let $(x, y) \in [0, 1]^2$ and suppose a SearchQT(u, (x, y)) request reached node v_k after $k \in \mathbb{N}_0$ hops, k even. Then the maximum Euclidean distance from v_k to the position (x, y) is at most $1/2^{(k-1)/2}$.*

Proof. Let $k \in \mathbb{N}_0$ be the number of hops until SearchQT(u, (x, y)) terminates. Assume that k is even. Let (u_x, u_y) be the coordinates of u. Initially the Euclidean distance between (u_x, u_y) and (x, y) is maximized if both coordinates lie on the corners of P such that the straight line between (u_x, u_y) and (x, y) is the diagonal going through P.

Note that after two hops we reduce the area in which the target is located by a factor $\frac{1}{4}$. Using the Pythagorean theorem to compute the length of the diagonal of

the quad, we compute the maximum distance between the node v_k and (x, y), to be equal to $\sqrt{(1/\sqrt{2^k})^2 + (1/\sqrt{2^k})^2} = 1/\sqrt{2^{k-1}} = 1/2^{(k-1)/2}$. □

We are now ready to prove the following theorem:

Theorem 8.19. *If the Euclidean distance $||uv||_2$ between any pair $(u, v) \in V$ is at least $1/n$, then any search message is delegated at most $\mathcal{O}(\log n)$ times.*

Proof. Assume that a SearchQT$(u, (x, y))$ message is at node v_k after k hops. It is easy to see that after each delegation, the remaining area in which we have to search for (x, y) is halved. We know by Lemma 8.18 that the maximum Euclidean distance from v_k to (x, y) within k hops is at most $1/2^{(k-1)/2}$ when k is even. Set $k = 4 \log n$. Then the maximum Euclidean distance is at most

$$\frac{1}{2^{(4 \cdot \log n - 1)/2}} = \frac{1}{2^{2 \cdot \log n - 1/2}} = \frac{\sqrt{2}}{2^{2 \cdot \log n}} = \frac{\sqrt{2}}{n^2} = \mathcal{O}\left(\frac{1}{n^2}\right) < \mathcal{O}\left(\frac{1}{n}\right),$$

which implies that the remaining area in which we have to search does not contain a node other than v_k, so the routing protocol terminates. As $k \in \mathcal{O}(\log n)$, the theorem follows. □

8.6. Self-Stabilizing Octrees

In this section we discuss how to extend our protocols to high-dimensional settings in order to support self-stabilizing octrees with geographic monotonic searchability. Fix a dimension $d > 2$: i.e., we are given a d-dimensional hypercube P of unit side length. Then each node u has coordinates $(u_1, \ldots, u_d) \in [0, 1]^d$.

We generalize the QuadDivision procedure as follows: Instead of alternating between two different cuts (vertical and horizontal cuts), we alternate between d different cuts now. Thus, for all $i \in \{1, \ldots, d\}$ we define an *i-cut* on the (sub-)cube P whose side length in dimension i is equal to I as follows. We assign all points $p \in P$ whose i-th coordinate is smaller than $\frac{1}{2}I$ to the subcube P_1, and the rest of the points to the subcube P_2. For an example consider Figure 8.4 for a sequence of different cuts on a 3-dimensional hypercube. By this, the QuadDivision algorithm remains well-defined.

Next, consider the area tree T that represents the output of the new QuadDivision algorithm. T again is an area tree. However, the levels of the area tree now alternate between d different cuts instead of only 2. Thus, we obtain the total ordering $\prec$ in the same manner as before, namely by performing a DFS on T, always going to the left child first. This already implies that BuildList is also well-defined in the d-dimensional setting.

Last but not least, it is easy to see that one can generalize the definition for $A(u)$ and $Q(u)$ (Definition 8.6) to dimension d, since the area tree T still is well-defined. This implies that we have a well-defined legitimate state according to the generalization of Definition 8.8 and thus the BuildQT protocol along with the routing protocol SearchQT is well-defined such that all claims made in the analysis can also be generalized to d-dimensional settings.

The following corollary summarizes the above discussion:

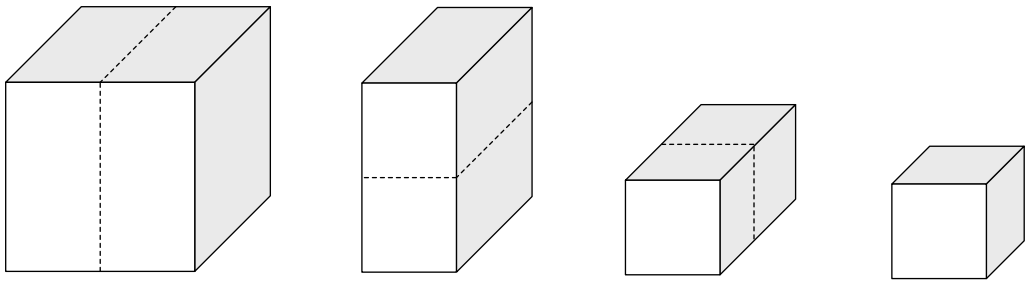


Figure 8.4.: Illustration of the 3-dimensional equivalent of QuadDivision. The sequence shows a 1-cut ((a) → (b)), followed by a 2-cut ((b) → (c)) and a 3-cut ((c) → (d)). The dashed lines indicate how the next cut in the sequence is applied to the (sub-)cube.

Corollary 8.20. *There exists a self-stabilizing protocol for a (d-dimensional) octree along with a routing algorithm $\mathcal{R}$ that satisfies geometrical monotonic searchability.*

It is also easy to see that the generalized version of SearchQT forwards a message at most $\mathcal{O}(\log n)$ times until termination in case the Euclidean distance (in the d-dimensional space, i.e., for points $x = (x_1, \ldots, x_d) \in [0,1]^d$ and $y = (y_1, \ldots, y_d) \in [0,1]^d$ we define $||xy||_d = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$) between any two nodes $u, v \in V$ is at least $1/n$. Finally, we want to emphasize that the variables for each node $u \in V$ do not change in our protocol, when applied to the higher-dimensional case.

CHAPTER 9

Self-Stabilizing Publish-Subscribe Systems

The publish-subscribe paradigm ([Eug+03; Fab+01]) is a very popular paradigm for the targeted dissemination of information. It allows clients to subscribe to certain topics or contents so that they will only receive information that matches their interests. In the traditional client-server approach the dissemination of information is handled by a server (sometimes also called *broker*), which has the benefit that the publishers are decoupled from the subscribers. The publisher does not have to know the relevant subscribers and the publisher and subscribers do not have to be online at the same time. However, in this case the availability of the publish-subscribe system critically depends on the availability of the server, and the server has to be powerful enough to handle the dissemination of the publish requests.

An alternative approach is to use a peer-to-peer system. However, if no commonly known gateway is available, the system cannot recover from overlay network partitions. In practice, peer-to-peer systems usually have a commonly known gateway since otherwise new peers may not be able to join the system through a peer that is currently in the system (and can therefore process the join request). In our supervised overlay network approach we assume that there is a commonly known gateway, called *supervisor*. The supervisor handles subscribe and unsubscribe requests but is not involved in the dissemination of publish requests, which are treated by the subscribers in a peer-to-peer manner. We are interested in realizing a *topic-based* supervised publish-subscribe system, which means that peers can subscribe to certain topics (that are usually relatively broad and predefined by the supervisor).

Topic-based publish-subscribe systems have many important applications. Apart from providing a targeted news service, they can be used for tasks such as to realize a group communication service [FLS01], which is considered an important building block for many other applications ranging from chat groups and collaborative working groups to online market places (where clients publish service requests), distributed file systems and transaction systems. To ensure the reliable dissemination of publish requests in a topic-based publish-subscribe system, we present a self-stabilizing supervised publish-subscribe system. This ensures that for any initial state (including overlay network partitions) eventually a legitimate state will be reached in which all subscribers of a topic know about all publish requests that have been issued for that topic. The overlay network that is formed by the subscribers in a legitimate state is a supervised skip ring, i.e., a ring with additional shortcuts. The skip ring has a diameter of $\mathcal{O}(\log n)$. We also show that the overhead for the supervisor in our system is very low. In fact, the message overhead of the supervisor is just a constant for subscribe and unsubscribe operations, and the supervisor has a low maintenance overhead in a legitimate state.

Underlying Publication. This chapter is based on the following publication:

M. Feldmann, C. Kolb, C. Scheideler, and T. Strothmann. "Self-Stabilizing Supervised Publish-Subscribe Systems". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, cf. [Fel+18].

Outline of This Chapter. We first introduce the skip ring topology along with the notion of the supervisor in Section 9.1. In Section 9.2 we give an overview on some related work in addition to the related work in Section 6.3. We formally describe our protocol BuildSR for a self-stabilizing supervised skip ring in Section 9.3. Finally, we show how to use BuildSR in order to construct a self-stabilizing supervised publish-subscribe system (Section 9.4).

9.1. Supervised Skip Rings

In this section we formally introduce the skip ring topology and the supervisor. We start with the skip ring topology.

We first define labels that we will assign to nodes in the skip ring:

Definition 9.1. *Let $x \in \mathbb{N}_0$ be some number with unique binary representation $(x_d \dots x_0) \in \{0,1\}^*$ (where $x_d = 1$), i.e., $x = \sum_{i=0}^{d} 2^i x_i$. Define the mapping $l : \mathbb{N}_0 \to \{0,1\}^*$ such that*

$$l(x) = (x_{d-1} \dots x_0 x_d).$$

The label *$x \in \mathbb{N}_0$ is then defined by $l(x)$. Denote by $|l(x)|$ the number of bits of $l(x)$.*

Intuitively, l takes the leading bit x_d of the binary string $(x_d \dots x_0)$ representing the input value x and moves x_d to the unit position, resulting in the binary string $(x_{d-1} \dots x_0 x_d)$. Note that the leading bit x_{d-1} of $l(x)$ is allowed to be 0 and that l is invertible. We map labels to values in $[0,1)$ as follows:

Definition 9.2. *Let $x \in \mathbb{N}_0$ and let $y = (y_1 \dots y_d) = l(x)$. Define the* real-valued representation *of $l(x)$ by the mapping $r : \{0,1\}^* \to [0,1)$ with*

$$r(y) = \sum_{i=1}^{d} \frac{y_i}{2^i}.$$

By the mapping r we obtain a total ordering $\prec$ of all labels; i.e., for two labels $l(x), l(y) \in \{0,1\}^*$ it holds that $l(x) \prec l(y)$ if and only if $r(l(x)) < r(l(y))$. We use $\prec$ in the following to define the skip ring:

Definition 9.3 (Skip Ring)**.** *A* skip ring *$SR(n)$ is a graph $G = (V, E_R \cup E_S)$ with n nodes. G is defined as follows:*

(a) *Each node $u \in V$ has a unique label denoted by $label(u) \in \{0,1\}^*$ with $l^{-1}(label(u)) < n$.*

(b) *$(u,v) \in E_R \Leftrightarrow (u,v)$ are consecutive in the ordering induced by $\prec$ when viewing the $[0,1)$-interval as a ring. Denote the edges in E_R as* ring edges*. For a node $u \in V$ denote v as u's* left ring neighbor *if $(u,v) \in E_R$ and either $v \prec u$ or $label(u) = 0$. Otherwise, if $(u,v) \in E_R$ then v is u's* right ring neighbor*.*

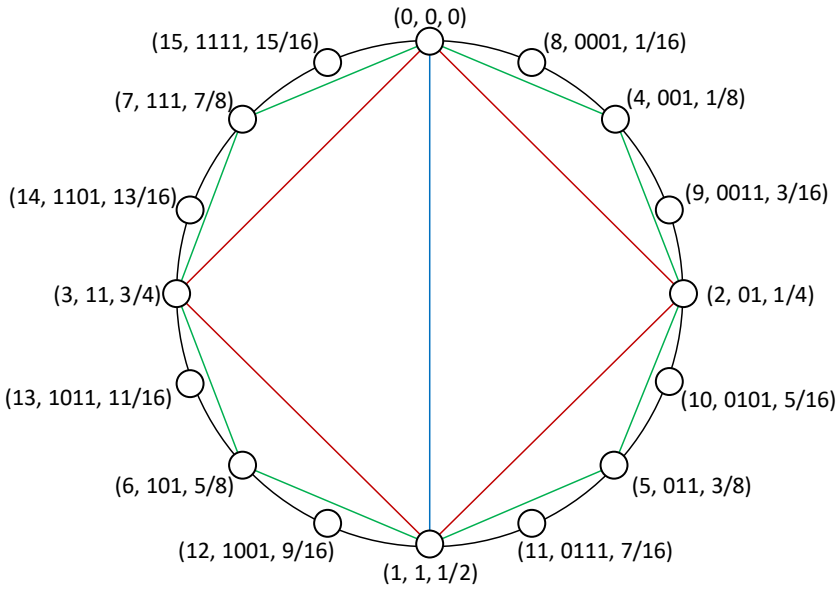


Figure 9.1.: A skip ring consisting of 16 nodes. The triples are of the form $(x, l(x), r(l(x)))$, where $x \in \{0, \ldots, 15\}$, $l(x)$ is the corresponding label and $r(l(x))$ is the real valued version of the label. Black edges are ring edges ($k = 4$), green edges are shortcuts for $k = 3$, red edges for level $k = 2$ and the blue edge is the shortcut for $k = 1$.

(c) $(u, v) \in E_S \Leftrightarrow (u, v)$ *is part of the sorted ring w.r.t. node labels over all nodes in* K_i, $i \in \{1, \ldots, \lceil \log n \rceil - 1\}$, *where* $K_i = \{w \in V \mid |label(w)| \leq i\}$. *Denote* $(u, v) \in E_S$ *as a* shortcut on level i, *if* $i = \max\{|label(u)|, |label(v)|\}$.

The intuition behind E_R and E_S is that we want all nodes with labels of length at most k to form a (bidirected) sorted ring for all $k \in \{1, \ldots, \lceil \log n \rceil\}$. For $k = \lceil \log n \rceil$ these edges are stored in E_R, for $k < \lceil \log n \rceil$ they are stored in E_S. Due to the way we defined the mapping l it holds that for all $x \in \{2^d, \ldots, 2^{d+1} - 1\}$ the values $r(l(x))$ are uniformly spread in between old values $r(l(y))$ with $y \in \{0, \ldots, 2^d - 1\}$. It is easy to see that the skip ring $SR(n)$ has the diameter $\lceil \log n \rceil$. Figure 9.1 illustrates $SR(16)$.

The following lemma follows from the definition of $SR(n)$:

Lemma 9.4 (Node Degree in $SR(n)$)**.** *In a skip ring* $SR(n)$ *the node degree is* $\mathcal{O}(\log n)$ *in the worst case and constant on average.*

Proof. For convenience, we define $k = |label(u)|$ for a node $u \in V$. Node u has 2 shortcuts to nodes with a label of length k' for each $k' \geq k$. Having n nodes in the system, we know that k' is upper bounded by $\log(n)$, which sums up the degree of u to be $2 \cdot (\log n - k + 1) = \mathcal{O}(\log n)$.

Next, we want to compute the average degree in $SR(n)$. We count the overall number of edges in $SR(n)$. Let $f(k)$ denote the number of nodes with a label of

length k. We have

$$f(k) = \begin{cases} 2 & k = 1 \\ 2^{k-1} & k > 1. \end{cases}$$

Recall that the maximum length of a label is equal to $\log(n)$ for $SR(n)$. Combining this fact with the above formula for the node degree, we get the following result for the number of edges in $|E_R \cup E_S|$:

$$\begin{aligned}
|E_R \cup E_S| &= \sum_{k=1}^{\log(n)} f(k)(2(\log(n) - k + 1)) \\
&= 4\log(n) + \sum_{k=2}^{\log(n)} 2^{k-1}(2(\log(n) - k + 1)) \\
&= 2\log(n) + \sum_{k=1}^{\log(n)} 2^{k}(\log(n) - k + 1) \\
&= 2\log(n) + \sum_{k=1}^{\log(n)} 2^{k}\log(n) - \sum_{k=1}^{\log(n)} 2^{k}k + \sum_{k=1}^{\log(n)} 2^{k} \\
&= 2\log(n) + (2n-2)\cdot\log(n) - \sum_{k=1}^{\log(n)} 2^{k}k + (2n-2) \\
&= 2\log(n) + (2n-2)\cdot\log(n) - (2n\log(n) - 2n + 2) + (2n-2) \\
&= 4n - 4.
\end{aligned}$$

Dividing this value by n yields an upper bound of $4 = \Theta(1)$ for the average node degree. □

As we will see, our protocols work in such a way that the node degree of a joining node u monotonically increases over the time u is part of the system; i.e., the longer a node is a participant of the system, the more shortcuts it has. This makes sense from a practical point of view, since older and thus more reliable nodes hold more connectivity responsibility in the form of more shortcuts. A node u that joins the system will be assigned a label such that its initial connections consist of two ring edges only. As further nodes join the system, u will receive new ring edges and transform its old ring edges into shortcuts, thus increasing its degree.

In order to construct a self-stabilizing protocol for a skip ring, we partially rely on the help of a gateway that is commonly known among all nodes. We represent such a gateway as a unique node, called the *supervisor*. The supervisor s is known to each node by default: i.e., we assume that each node $u \in V$ has a hard-coded edge (u, s) in any state of the system. We are now ready to define the topology that should be formed by nodes in legitimate states:

Definition 9.5 (Supervised Skip Ring)**.** *A* supervised skip ring *is a graph* $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$ *with* $n = |V|$ *nodes and a supervisor node* s*. G has the following properties:*

(a) *The subgraph* $G' = (V, E_R \cup E_S)$ *is a skip ring* $SR(n)$*, where* $n = |V|$.

(b) E_{SUP} *consists of edges* (u, s) *and* (s, u) *for each node* $u \in V$.

Note that while the edges from nodes to the supervisor are hard-coded and thus existent in every state of the system, the edges from the supervisor to the nodes are not hard-coded. This means that the supervisor only knows a subset of the nodes in V in an illegitimate state. Obviously, the bounds on the worst-case and average-case degree from Lemma 9.4 still hold for nodes in V in a supervised overlay network, since each node only has one additional outgoing edge. Having the supervisor store information about all nodes in the skip ring is acceptable for our setting, since storing information of multiple millions of nodes takes only a few Megabytes of storage for the supervisor. In addition, we are going to prove that the supervisor only receives one message in legitimate states after all nodes have called their Timeout action exactly once.

9.2. Related Work

We give an overview on related work specifically relevant to supervised overlay networks, self-stabilizing publish-subscribe systems and group communication services.

Supervised Overlay Networks. Supervised overlay network for specific topologies have been proposed in [PRU03; RS04a; RS04b]. In [PRU03] the nodes are arranged in a random-looking graph that guarantees connectivity, low diameter and low degree with high probability. A supervised tree called SPON is presented in [RS04b]. Such a tree is used to efficiently broadcast or multicast messages to a dynamically changing group of nodes. In [RS04a] a distributed hash table (DHT) based on a supervised De Bruijn graph is presented. The DHT distributes objects uniformly among all nodes and is designed to be useful in grid computing. A general framework for constructing a supervised peer-to-peer system has been introduced by Kothapalli and Scheideler [KS05]. Here the supervisor only has to store a constant amount of information about the system at any time and only has to send out a constant number of messages to integrate or remove a node. While all of the above mentioned systems have their advantages, none of them is self-stabilizing.

Self-Stabilizing Publish-Subscribe Systems. In the literature there are publish-subscribe systems that are self-stabilizing. For example, in [Müh+05] the authors present different content-based routing algorithms in a self-stabilizing (acyclic) broker overlay network that clients can publish messages to. Their main idea is a leasing mechanism for routing tables such that it is guaranteed that once a client subscribes to a topic there is a point in time such that every publication issued thereafter is delivered to the newly subscribed client (i.e., there are no guarantees for older publications). While the authors focus on the routing tables and take the overlay network as a given ingredient, our work focuses on constructing a self-stabilizing supervised overlay network and then using it to obtain a self-stabilizing publish-subscribe system.

A self-stabilizing publish-subscribe system for wireless ad-hoc networks is proposed in [ST18], which builds upon the work of [ST16b; STM15]. Similarly to our work, the authors arrange nodes in a cycle with shortcuts and present a routing algorithm

that makes use of these shortcuts to deliver new publications for topics to subscribers only after $\mathcal{O}(n)$ steps. Subscribe and unsubscribe requests are processed by updating the routing table at nodes. Both systems described above differ from our approach, as they solely focus on the routing scheme and updates of the routing tables, while we focus on updating the topology upon subscribe/unsubscribe requests. Our system is also able to deliver publications in $\mathcal{O}(\log n)$ steps if we use flooding, since we use a network with logarithmic diameter. Furthermore, we are also able to deliver all publications of a domain to a new subscriber after only a constant number of rounds.

Group Communication Services. There is a close relationship between group communication services (e.g., [FLS01; Ami+05]) and publish-subscribe systems. Nodes are ordered in groups in both paradigms and group-messages are only distributed among all members of some group. Self-stabilizing group communication services are proposed in [DSW06] for ad-hoc networks and in [DS04] for directed networks. However, there are some key differences: In group communication services, participants have to agree on group membership views. This results in a high memory overhead for each member of a group, as nodes in a group technically form a clique. On the other hand, subscribers of topics in publish-subscribe systems are in general not interested in any other members of the topic. For our approach, this results in a logarithmic worst-case and a constant average case degree for subscribers.

9.3. Self-Stabilizing Supervised Skip Rings

In this section we describe and analyze our self-stabilizing protocol for a supervised skip ring, called BuildSR. BuildSR consists of a protocol that is executed by the supervisor s (the *supervisor protocol*) and a protocol that is executed by each node $u \in V$ (the *subscriber protocol*).

9.3.1. Supervisor Protocol

The first part of the BuildSR protocol is executed by the supervisor. The supervisor maintains a database that is defined as follows:

Definition 9.6 (Supervisor Database)**.** *The supervisor s maintains a database $s.DB \subset \{0,1\}^* \times V$ containing labels corresponding to nodes.*

The task for the supervisor is to periodically inform each node in the skip ring of its correct label and its ring neighbors. We call this information of a node the *configuration*.

Definition 9.7 (Node Configuration)**.** *Let $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$ be a supervised skip ring and let $u \in V$. The* configuration $C(u)$ *for u is given by the database entries*

$$(label(v), v), (label(u), u), (label(w), w) \in s.DB,$$

where $v \in V$ is u's left neighbor and $w \in V$ is u's right neighbor in the ring formed by edges E_R.

The supervisor protocol (Algorithm 18) consists of a Timeout action that is periodically executed by the supervisor s, an action GetConfiguration with parameter $u \in V$ that lets s send $C(u)$ to u according to $s.DB$ and an action Subscribe with parameter $u \in V$ that introduces u to s.

Algorithm 18 The supervisor protocol, executed by the supervisor s

```
1:  Timeout → true
2:      Check integrity of s.DB
3:      Pick (label, u) ∈ s.DB in a round-robin fashion
4:      GetConfiguration(u)

5:  GetConfiguration(u)
6:      Check integrity of s.DB
7:      if ∃(label(v), v) ∈ s.DB : v = u then
8:          Let C(u) be u's configuration according to s.DB
9:          v ← SetData(C(u))
10:     else
11:         u ← SetData(⊥)

12: Subscribe(u)
13:     Check integrity of s.DB
14:     if ∀(label(v), v) ∈ s.DB : u ≠ v then
15:         s.DB ← s.DB ∪ (l(u), u)
16:     GetConfiguration(u)
```

In Timeout, the supervisor s chooses a pair $(label(u), u) \in s.DB$ in a round-robin fashion and then locally calls GetConfiguration for u. Picking a pair in round-robin fashion guarantees that after the supervisor has been activated $|s.DB|$ times, each pair that is contained in the database has been picked exactly once.

In GetConfiguration, s first checks if its database contains a tuple that contains the node u. If this is the case, s can locally compute $C(u)$ using $s.DB$ and send a message to u containing $C(u)$. This results in u executing the action SetData, which will be explained in the subscriber protocol. If s cannot find u in its database, it replies to u by calling SetData with the parameter $\perp$ on u.

If a node u introduces itself to s by calling Subscribe(u) on s, then, in case there is no entry in $s.DB$ that contains u, s generates a new label $l(u)$ for u and adds the entry $(l(u), u)$ to $s.DB$. In any case, s replies to u by sending u its configuration $C(u)$ via GetConfiguration(u).

In addition to the above actions, the supervisor has to check the integrity of its database: i.e., s has to check that each node u has a correct label associated to it according to Definition 9.3. Also, s has to check that each node u is only present in exactly one tuple $(label(u), u)$ in $s.DB$ and that all labels are unique. If there are inconsistencies as described above detected by s, then s can easily recompute its database locally such that it is consistent afterwards without removing node references. We therefore assume that the $s.DB$ is always in a consistent state from this point on.

9.3.2. Subscriber Protocol

In this section we discuss the part of the BuildSR protocol that is executed by each node $u \in V$. First, we present the variables needed for a node. Note that we intentionally omit the reference to the supervisor s here, since links to s are assumed to be hard-coded.

Definition 9.8. *For the BuildSR protocol, each node $u \in V$ maintains the following variables:*

(a) $u.label \in \{0,1\}^* \cup \{\perp\}$*: u's unique label or* $\perp$ *if u has not received a label yet.*

(b) $u.left, u.right \in (\{0,1\}^* \times V) \cup \{\perp\}$*: u's left and right ring neighbor.*

(c) $u.shortcuts \subset \{0,1\}^* \times V$*: u's shortcut connections.*

Using the variables $u.left$ and $u.right$, all nodes $u \in V$ aim to form a sorted ring in a self-stabilizing manner. This can be done via the BuildRing protocol from Section 6.4, where the ordering $\prec$ is defined via the real-valued representation of the node labels (see Definition 9.2). Note that in order for BuildRing to work here, each node u that introduces itself to some node v via the Linearize action only does so if $u.label \neq \perp$. Node u then has to send its node reference and its current label $u.label$ to v, as otherwise v could not locally compare its own label $v.label$ to $u.label$. Also, upon executing Linearize for the parameters u and $u.label$, v first checks if u is already stored in either $v.left$ or $v.right$. If that is the case, say $v.left = (label_u, u)$, then v replaces $label_u$ by $u.label$ first in order to keep its knowledge of u's label up to date.

Unfortunately, it cannot be guaranteed in initial states that each node u already has a correct label assigned to $u.label$. Therefore, we describe a mechanism in the first part of the subscriber protocol that eventually lets each node receive its correct label from the supervisor. This automatically implies that the supervisor has references to all nodes stored in its database by this time. The second part of the subscriber protocol then deals with establishing the necessary shortcuts for each node u. The pseudocode for the subscriber protocol is given in Algorithm 19.

Receiving Correct Labels. For now, we focus on the ring edges only. Our first goal is to guarantee that every node u eventually stores its correct label in $u.label$.

Recall that we have periodic communication from the supervisor to the nodes: i.e., the supervisor periodically sends out the configurations to all nodes $u \in V$ stored in its database. This action alone does not suffice to make sure that every node eventually stores its correct label, since in initial states the supervisor's database may be empty and node labels may store arbitrary values. Thus, we also need periodic communication from nodes to the supervisor. The challenge here is to not overload the supervisor with requests in legitimate states of the system. Each node u periodically executes the following actions:

(i) If $u.label = \perp$, then u asks the supervisor to integrate u into the *database* and send u its correct configuration by calling Subscribe(u).

(ii) When u receives its configuration $C(u)$ from the supervisor, it does the following: If $C(u) = \perp$, then u got notified by the supervisor that u is not contained in

Algorithm 19 The BuildSR protocol, executed by nodes $u \in V$

```
1:  Timeout → true
2:      if u.label =⊥ then
3:          s ← Subscribe(u)
4:      else if u's label is minimal among all of u's neighbors then
5:          s ← GetConfiguration(u)
6:      if ∃(label_v, v), (label_w, w) ∈ u.shortcuts on level k = |u.label| then
7:          v ← IntroduceShortcut(label_w, w)
8:          w ← IntroduceShortcut(label_v, v)

9:  SetData(C(u))
10:     if C(u) =⊥ then
11:         u.label ←⊥
12:     else
13:         Let C(u) = ((label(v), v), (label(u), u), (label(w), w))
14:         u.label ← label(u)
15:         if u.left ≠⊥ ∧u.left ≠ (label(v), v) then
16:             s ← GetConfiguration(u.left)
17:         if u.right ≠⊥ ∧u.right ≠ (label(w), w) then
18:             s ← GetConfiguration(u.right)
19:         u.left ← (label(v), v)
20:         u.right ← (label(w), w)

21: IntroduceShortcut(label_v, v)
22:     if ∃(label_w, w) ∈ u.shortcuts : l' = l then
23:         if w ≠ v then
24:             u.shortcuts ← u.shortcuts \ {(label_w, w)}
25:             Linearize((label_w, w))
26:         u.shortcuts ← u.shortcuts ∪ {(label_v, v)}
27:     else
28:         Linearize((label_v, v))
```

the database. Consequently, u resets its label to $\perp$. Otherwise u updates its label and checks if the left and right ring neighbors indicated by $C(u)$ match its variables $u.left$ and $u.right$. If, for example, $u.left$ does not match the left ring neighbor in $C(u)$, u requests the supervisor to send $u.left$ its correct configuration and replaces $u.left$ by the left ring neighbor indicated by $C(u)$.

(iii) u periodically requests its configuration from the supervisor if it determines, only on the basis of its local information, that its label is minimal.

As it will turn out in the analysis, these actions together with the extension for the BuildRing protocol described above suffice to guarantee that eventually each node u receives its correct label and gets stored in the supervisor's database.

Establishing Shortcut Connections. In this section we describe how the nodes establish and maintain shortcut edges. Recall that shortcuts are on levels $k =$

$\{1, \ldots, \lceil \log n \rceil\}$ (Definition 9.3), where $k = \lceil \log n \rceil$ represents the ring edges that are already established. A node u with a label of length $k = |u.label|$ has exactly 2 shortcuts on each level in $k, \ldots, \lceil \log n \rceil$ in a legitimate state.

We first describe how a node is able to compute all its shortcut labels locally, based only on the information of its left and right ring neighbors. The following approach only computes the respective labels in $[0, 1)$ that a node should have shortcuts to, but not the nodes that are associated with these labels. The idea is the following. In general, a node $u \in V$ has shortcuts only to other nodes that lie on the same semicircle as u, i.e., either the semicircle of nodes within the interval $[0, 1/2]$ or the semicircle of nodes within the interval $[1/2, 1]$ (where the 1 is represented by the node with the label 0). Consider a subscriber u with $r(u.label) \in [0, 1)$ and its two ring neighbors v, w such that $u.left = (label_v, v)$ and $u.right = (label_w, w)$. If u recognizes that $|u.label| < |label_v|$, then u knows that it has to have a shortcut with the label l and $r(l) = 2 \cdot r(label_v) - r(u.label)$, because node v was previously inserted between the nodes with labels l and $u.label$. After this, u can apply this method recursively: i.e., it checks for the computed label l if $|u.label| < |l|$ until it reaches a label of less or equal length. This same procedure is applied analogously for $u.right$.

The following example illustrates our approach:

Example 9.9. *Recall the skip ring from Figure 9.1. Suppose we want to compute all shortcut labels for the node with (real-valued) label $\frac{1}{4}$, only on the basis of the labels of its direct ring neighbors, which are $\frac{3}{16}$ and $\frac{5}{16}$. We know that the label $\frac{3}{16}$ has the length 4, which is greater than the length of label $\frac{1}{4}$, which is 2. Thus, we get a shortcut s_1 for $\frac{1}{4}$ with the label $2 \cdot \frac{3}{16} - \frac{1}{4} = \frac{1}{8}$. The label $\frac{1}{8}$ has the length 3, which is still greater than 2. Hence, we compute a shortcut s_2 with the label $2 \cdot \frac{1}{8} - \frac{1}{4} = 0$. Finally, we know that the length of label 0 is 1, which is smaller than 2, which terminates the algorithm. The computation of shortcut labels to $\frac{3}{8}$ and $\frac{1}{2}$ works analogously.*

We are now ready to describe the self-stabilizing protocol that establishes and maintains shortcuts for all nodes. Consider a node u with the label length $|u.label| = k$. On Timeout, u checks if $u.shortcuts$ contains nodes $(label_v, v), (label_w, w)$ on level k. If that is the case, then u introduces v to w by sending a message to w containing the reference of v as well as v's label $label_v$. Also, u introduces w to v in the same manner. Note that for $|u.label| = \lceil \log n \rceil$, u has to consider its two ring neighbors instead of $u.shortcuts$. On receipt of such an introduction message consisting of the pair $(label_w, w)$, v checks if it has a shortcut $(label_{w'}, w')$ with $label_{w'} = label_w$. If that is the case, then v replaces the existing node reference w' by w and, if $w' \neq w$, forwards the reference of w' to the sorted ring via the BuildRing protocol. This way it is guaranteed that shortcuts are established in a bottom-up fashion.

9.3.3. Analysis

In this section we show that BuildSR is self-stabilizing according to Definition 9.3.

First of all, note that eventually all corrupted messages that may exist in an initial state are received and processed. Furthermore, a corrupted message cannot trigger an infinite chain of corrupted messages:, i.e., eventually the false information is either

corrected or received but not spread further anymore. We assume this fact for the rest of the proof.

We start by proving that eventually the supervisor has all nodes contained in its database. For this, call a node $u \in V$ *recorded* if there exists $(label(u), u) \in s.database$.

Lemma 9.10 (Supervisor Convergence)**.** *Eventually all nodes $u \in V$ are recorded.*

Proof. Note that the supervisor s does not remove nodes from its (non-corrupted) database, as it is able to locally restore the database from an initially corrupted state without dropping any node references.

Let $u \in V$ be some node that is not yet recorded. For now, assume that $u.label = \perp$. Then u requests its configuration from s by calling Subscribe on s in its Timeout action and becomes recorded afterwards.

It remains to consider the general case where we are given a connected component $C \subset V$ of nodes, where each node has a label not equal to $\perp$. Since we run the BuildRing protocol, the nodes in C eventually form a sorted ring according to their labels. It follows that C eventually contains at least one node that is recorded, i.e., the node with minimal label. As long as the supervisor is able to introduce new recorded nodes to nodes already recorded in C, C's size grows. But since the number of nodes is finite, C will eventually become static. We show that for such a static connected component C, eventually all nodes in C will become recorded. Consider the potential function

$$\Phi(C) = |\{u \in C \mid u \text{ is non-recorded}\}|.$$

We show that eventually, $\Phi(C) = 0$. Since the supervisor does not drop connections to nodes, $\Phi(C)$ is never increasing. Let $\Phi(C) = c$ for an arbitrary integer $c > 0$. Then all nodes in C eventually form a sorted ring due to BuildRing. This implies that there exists a ring edge (u, v) from a node u that is already recorded to a subscriber v that is not yet recorded. W.l.o.g. let $u.right = v$. Since v is not recorded, u's configuration according to $s.database$ has to contain a different right neighbor than v. Let $w \in C, w \neq v$ be this neighbor: i.e., as the supervisor sends u its correct configuration, it tells u that w should be its right ring neighbor. But then it has to hold $label(u) \prec label(v) \prec label(w)$. This implies that upon receiving its configuration from the supervisor, u requests the configuration for v at the supervisor, leading to v changing its label to $\perp$ and thus getting recorded afterwards due to the above argumentation. Hence, $\Phi(C)$ is reduced by one and is therefore monotonically decreasing. □

Having the supervisor's database converged, we know that the ring of all nodes eventually converges:

Lemma 9.11 (Ring Convergence)**.** *Once each node $u \in V$ has been recorded, the ring induced by edges E_R eventually converges.*

Proof. The supervisor periodically sends the correct configuration to each node $u \in V$ in a round-robin fashion. This implies that after n calls of the supervisor's Timeout action, each node has stored its correct label. Note that this does not necessarily include the correct ring neighbors right away: A node $u \in V$ may have received

its configuration $C(u) = (pred_u, label_u, succ_u)$ from the supervisor, but the node v stored via $pred_u$ or $succ_u$, respectively, may not. This results in u modifying $u.left = pred_v$ via BuildRing, because v does not have received its correct label yet. Since now all labels are correct, by the time each node u has received its configuration from the supervisor again, u does not change its list neighbors anymore. □

Finally, we need to prove the convergence of the shortcuts for all subscribers:

Lemma 9.12 (Shortcut Convergence)**.** *Once each node $u \in V$ has been recorded and the sorted ring has converged, all edges in E_S will eventually be established.*

Proof. We perform an induction over the levels $i = \lceil \log n \rceil, \ldots, 1$ of shortcuts and show that all shortcuts on each level are eventually established. The induction base ($i = \lceil \log n \rceil$) trivially holds, as shortcuts on level $\lceil \log n \rceil$ are ring edges in E_R. For the induction hypothesis, assume that all shortcuts on level i have already been established, i.e., all nodes in $K_i = \{v \in V \mid |label(v)| \leq i\}$ already form a sorted ring (recall Definition 9.3). In the induction step we show that all shortcuts on level $i-1$ are eventually established. It is easy to see that $K_{i-1} \subset K_i$ holds. Denote the sorted ring over nodes in K_i as R_i. Observe that each node $u \in K_i \setminus K_{i-1}$ has two neighbors v, w in R_i with $v, w \in K_{i-1}$. Thus, by definition of our protocol, u eventually introduces v to w and vice versa via the action IntroduceShortcut in its Timeout action. This implies that the shortcuts (v, w) and (w, v) are established. The above argumentation implies that the ring R_{i-1} is established eventually, which concludes the induction. □

Having shown the convergence of the supervisor (Lemma 9.10), the sorted ring for all nodes (Lemma 9.11) and the convergence of the shortcuts for all nodes (Lemma 9.12), we obtain the following lemma:

Lemma 9.13 (Convergence)**.** *Given any initially weakly connected graph $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$, BuildSR transforms G into a supervised skip ring.*

It remains to show the closure property for BuildSR.

Lemma 9.14 (Closure)**.** *If the explicit edges in $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$ already form a supervised skip ring, then they are preserved at any point in time if no nodes join or leave the system.*

Proof. We need to show closure for the supervisor's database as well as for the skip ring. As already argued, the supervisor does not drop node references stored in its database, so closure for the supervisor follows trivially.

Messages that are generated by the BuildRing protocol do not modify the edge set E_R, since closure of BuildRing (Theorem 6.14) holds. Observe that introduction messages for shortcuts do not modify the variables $u.left$ and $u.right$ for a node $u \in V$. Implicit edges generated by configurations sent out by the supervisor s are just merged with the existing explicit edges at the receiving node u, since u already stores the correct configuration.

Note that shortcuts are only modified via the action IntroduceShortcut. IntroduceShortcut is only called to introduce a node v to some shortcut w, which already

exists, since no node generates an introduction message for two nodes that are not allowed to be connected by a shortcut. □

By combining Lemmas 9.13 and 9.14, we get the main result of this section:

Theorem 9.15. *BuildSR is self-stabilizing.*

One can easily conclude the following theorem bounding the number of messages the supervisor receives in legitimate states once each node has executed its Timeout action exactly once. This means that the load on the supervisor is low in legitimate states.

Theorem 9.16. *Consider a supervised skip ring $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$ in a legitimate state. If each node $u \in V$ executes its Timeout action exactly once, the supervisor receives one GetConfiguration message.*

Proof. Only the node u with a minimal label sends a GetConfiguration message to the supervisor, see Algorithm 19. □

9.4. Self-Stabilizing Publish-Subscribe Systems

In this section we show how to use the BuildSR protocol as a (topic-based) self-stabilizing publish-subscribe system. Let $\mathcal{T} \subset \mathbb{N}$ be a set of integers, where each $t \in \mathcal{T}$ represents a *topic*.

A node u is allowed to issue the following requests to the supervisor:

- Subscribe(u, t): u subscribes to the topic t.
- Unsubscribe(u, t): u unsubscribes from the topic t.

A node u that is subscribed to the topic t is called a *subscriber* for t.

Additionally u may publish a new message via the following request:

- Publish(m, t): Delivers the message $m \in \{0,1\}^*$ to all subscribers for t.

Note that we do not need the supervisor to be involved in the execution of Publish requests.

To construct a publish-subscribe system out of our self-stabilizing supervised overlay network, we basically run a BuildSR protocol for each available topic $t \in \mathcal{T}$ at the supervisor. Thus, the supervisor has to extend its database to be in $\{0,1\}^* \times \mathcal{T} \times V$: i.e, each node u has a separate label for each topic it is subscribed to. Each node u also runs a separate BuildSR protocol for each topic it is subscribed to. Once a node wants to subscribe to some topic $t \in \mathcal{T}$, it starts running a new BuildSR protocol for topic t. Upon unsubscribing, the node removes the corresponding BuildSR protocol and ignores all incoming messages that correspond to the topic t once it gets the permission from the supervisor to do so. The above extensions can be implemented easily by having each message generated in the BuildSR protocol for the topic t now contain t as a parameter.

9.4.1. Protocol Description

In this section, we describe the protocols for the requests Subscribe, Unsubscribe and Publish. Note that we present two separate protocols for Publish. One is simple, fast and follow a basic flooding approach. The other one is self-stabilizing: i.e., it guarantees that eventually all subscribers store all publications for the topics they subscribed to.

Subscribe

Upon receiving a Subscribe(u, t) request, the supervisor basically behaves the same as in the Subscribe action of Algorithm 19: i.e., it locally updates its database by adding a new entry $(l_t(u), t, u)$. Here $l_t(u)$ denotes the label of u in the skip ring of all subscribers for the topic t. Once the supervisor s has updated its database, s sends u its configuration in the skip ring for the topic t.

The way the supervisor assigns labels to subscribes of some topic has the advantage that it spreads multiple sequential Subscribe requests through the skip ring, meaning that a pre-existing subscriber is involved (i.e., it has to change its configuration) only for two consecutive Subscribe requests. Afterwards, its configuration remains untouched until the number of subscribers has doubled. This is due to the definition of the label function l (recall Definition 9.1).

Example 9.17. *Consider the skip ring* $SR(16)$ *from Figure 9.1 and assume that there are* 16 *new nodes that want to subscribe. Then these new nodes are inserted in between consecutive pairs of (old) subscribers on the ring, as they receive (real-valued) labels* $1/32$, $3/32$, $5/32, \ldots, 31/32$.

Unsubscribe

Let V_t be the set of nodes that are subscribed to the topic $t \in \mathcal{T}$ and denote by $C_t(u)$ the configuration of node u with respect to the skip ring formed by all subscribers for the topic t. When processing an Unsubscribe(u, t) request, the supervisor executes the actions specified by Algorithm 20:

Algorithm 20 Unsubscribe(u, t) handled by the supervisor s

1: Get the entry $(l_t(v), t, v)$ with $l_t(v) = l_t(|V_t| - 1)$ from $s.database$ and replace $l_t(v)$ by $l_t(u)$ in $s.database$.
2: Remove $(l_t(u), t, u)$ from $s.database$.
3: Send v's old ring neighbors, v's new ring neighbors and v their updated configuration $C_t(v)$.
4: Send $C_t(u) = \perp$ to u.

Note that the supervisor's database is already in a legitimate state after the Unsubscribe request has been processed. Therefore, the supervisor does not rely on additional information from other subscribers to maintain its database. Furthermore, note that the supervisor only has to send out a constant amount of configurations per Unsubscribe request: two configurations for v's old ring neighbors, two for v's

new ring neighbors, one for v itself and one for u. The shortcut connections are then updated automatically by BuildSR. Once u has received its configuration it removes the corresponding BuildSR protocol and ignores all incoming messages that correspond to the topic t. As u is now excluded from the skip ring, it is easy to see that u eventually does not receive any more messages regarding the topic t anymore.

Publish

For simplicity we only consider the skip ring for a fixed topic $t \in \mathcal{T}$ for the description of the following protocol. Spreading publications among all subscribers is done through flooding. We also present a separate protocol which is self-stabilizing in a sense that the protocol ensures that eventually all subscribers store all publications. The self-stabilizing protocol for publications is able to correct eventual mistakes that occurred in the flooding approach. For storing publications at each subscriber, we use an extended version of a Patricia trie [Mor68] to effectively determine missing publications at subscribers.

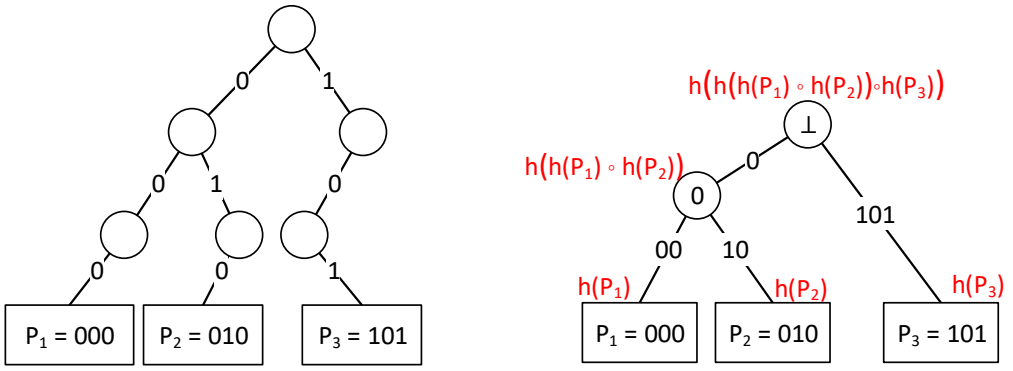
Definition 9.18 (Trie)**.** *A* trie *is a search tree with node set T over the alphabet $\Sigma = \{0, 1\}$. Every edge is associated with a label $c \in \Sigma$. Additionally, every key $x \in \Sigma^k$ that has been inserted into the trie can be reached from the root of the trie by following the unique path of length k whose concatenated edge labels result in x.*

Definition 9.19 (Patricia trie)**.** *A* Patricia trie *is a compressed trie in which all chains (i.e., maximal sequences of nodes with only one child) are merged into a single edge whose label is equal to the concatenation of the labels of the merged trie edges.*

Each subscriber $u \in V$ maintains a Patricia trie, denoted by $u.T$. Each leaf node in a Patricia trie stores a publication $p \in \{0, 1\}^*$. Each inner node $t \in T$ of a Patricia trie has exactly 2 child nodes denoted by $c_1(t), c_2(t) \in T$. Furthermore, we assign a label to each node. The label $t.label \in \Sigma^k$ of a leaf node $t \in T$ is just equal to the publication p stored by t. The label $t.label \in \Sigma^k$ of an inner node $t \in T$ is defined as the longest common prefix of the labels of t's child nodes (with $\perp$ being the empty word).

In addition to node labels, we assign a hash value to each node in the Patricia trie: We use a publicly known pseudorandom collision-resistant hash function $h : \{0, 1\}^* \to \{0, 1\}^*$ and define the hash value $t.hash$ of a leaf node t as $h(t.label)$. If t is an inner node, then $t.hash$ is defined as the hash of the concatenation of the hashes of t's child nodes: i.e., $t.hash = h(h(c_1(t)) \circ h(c_2(t)))$. This construction is similar to a Merkle-Hash Tree (MHT) [Mer87]. It is easy to see that if a subscriber $u \in V$ stores a set of publications P, u can locally compute the corresponding Patricia trie for P without having to communicate with other subscribers. Consider Figure 9.2 for a trie and its corresponding Patricia trie.

Flooding. We make use of the skip ring's shortcuts to spread new publications over the ring. Whenever a subscriber $u \in V$ generates a new publication p, u inserts p into $u.T$ and broadcasts p over the skip ring, by sending a Flood(p) message to all of its neighbors v with $(u, v) \in E_R \cup E_S$. Upon receiving such a Flood(p) message, a subscriber $v \in V$ checks if it has already stored p in $v.T$. If not, then v inserts p into



(a) Trie (b) Patricia trie

Figure 9.2.: A trie (a) and its corresponding Patricia trie (b) storing publications P_1, P_2, P_3. Red node labels in (b) denote the hash values for the nodes.

$v.T$ and continues to broadcast p by forwarding the Flood message to its neighbors. In case p is already stored in $v.T$, v just drops the message. Algorithm 21 states the pseudocode for the flooding approach.

Algorithm 21 Flooding performed at node $u \in V$

```
1: Flood(p)
2:     if p is not stored in u.T then
3:         Insert p into u.T
4:         for all v ∈ {u.left, u.right} ∪ u.shortcuts
5:             v ← Flood(p)
```

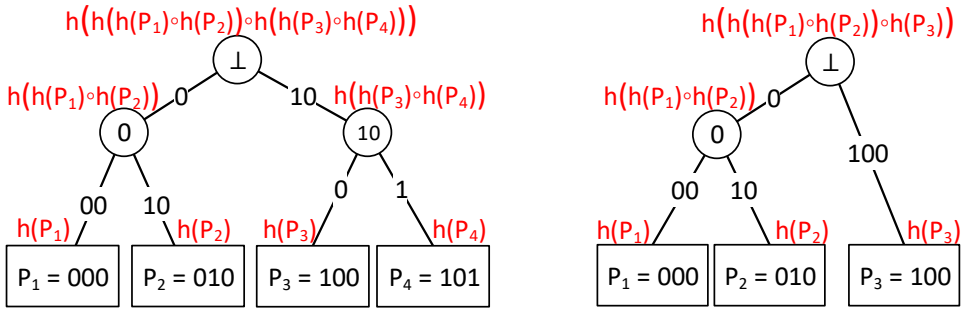
By applying this flooding approach, it only takes $\lceil \log n \rceil$ hops for a publication to reach any subscriber in the skip ring (recall that the skip ring has the diameter $\lceil \log n \rceil$).

Self-Stabilizing Publications. We now describe a self-stabilizing protocol that ensures that all subscribers eventually store all publications in their Patricia tries. To ease presentation we assume that the system is in a state where each subscriber $u \in V$ stores a subset $P_u \subseteq P$ of publications in $u.T$ from a set P of publications such that $\bigcup_{u \in V} P_u = P$. Our protocol guarantees that eventually each subscriber u stores all publications in P in $u.T$. The protocol consists of a Timeout action that is periodically executed and 3 actions Receive, CheckTrie and CheckAndReceive.

In Timeout u periodically sends a request CheckTrie(u, $r.label$, $r.hash$) to one of its ring neighbors (chosen alternately) containing u itself as well as the label $r.label$ and the hash value $r.hash$ of the root node r of $u.T$.

Upon receiving a request CheckTrie(v, l_v, h_v), a subscriber $u \in V$ does the following: It searches for the node $t_u \in u.T$ with the label $t_u.label = t_v.label$ and checks if $t_u.hash = t_v.hash$. The following three cases may occur:

(i) $t_u.hash = t_v.hash$: Then u knows that the set of publications stored in the subtree of $u.T$ with root node t_u are the same as the set of publications stored



(a) $u.T$ (b) $v.T$

Figure 9.3.: Example Patricia tries $u.T$ and $v.T$ for two subscribers $u, v \in V$.

in the subtree of $v.T$ with root node t_v. Consequently, u does not send any response to v in this case.

(ii) $t_u.hash \neq t_v.hash$: Then the contents of the subtrees with roots t_u, t_v differ in at least one publication. In order to detect the exact location, where both Patricia tries differ, u responds to v by sending requests CheckTrie(u, $c_1(t_u).label$, $c_1(t_u).hash$) and CheckTrie(u, $c_2(t_u).label$, $c_2(t_u).hash$) to v.

(iii) There is no node $t_u \in u.T$ such that $t_u.label = t_v.label$. Then $v.T$ contains publications that do not exist in $u.T$. Node u computes the label prefix of those missing publications as follows. First, u searches for the node $t'_u \in u.T$ with the label prefix $t_v.label$ and $|t'_u.label|$ minimal: i.e., $t'_u.label = t_v.label \circ b_1 \circ \ldots \circ b_k$ with $b_1, \ldots, b_k \in \{0,1\}$ and $|t'_u.label| = |t_v.label| + k$ minimal. If such a node t'_u exists, then $u.T$ may contain at least all publications with the label prefix $t'_u.label$. Furthermore, u knows that all publications with the label prefix $t_v.label \circ (1 - b_1)$ are missing in $u.T$. Therefore, u requests v to continue checking the subtree with root node of label $t'_u.label$ and to deliver all publications with the label prefix $p = t_v.label \circ (1 - b_1)$ to u. It does so by sending a CheckAndReceive(u, $t'_u.label$, $t'_u.hash$, p) request to v. Let P denote the set of these publications that v has to deliver to u. Upon executing the CheckAndReceive request, v internally calls CheckTrie(u, $t'_u.label$, $t'_u.hash$) and, in addition, delivers P by sending a Publish(P) request to u. In case a node t'_u as described above cannot be found in $u.T$, u just requests v to deliver all publications with the prefix $t_v.label$ to u, since that entire subtree is missing in $u.T$.

With this approach, the only publications that are send out are those that are missing at the receiver. Algorithm 22 states the pseudocode for our protocol.

Example 9.20. *Consider two subscribers $u, v \in V$ with Patricia tries as shown in Figure 9.3. Note that P_4 is missing in $v.T$. We describe how v will eventually receive P_4 from u when using Algorithm 22.*

First assume that u sends out a CheckTrie(u, $r_u.label$, $r_u.hash$) message to v in its Timeout action, with r being the root node of $u.T$. Node v then compares $r_u.hash$ with the hash $r_v.hash$ of its root node $r_v \in v.T$. Since $r_u.hash \neq r_v.hash$, v sends requests

Algorithm 22 Self-Stabilizing publication protocol executed by subscriber $u \in V$

```
1: Timeout → true
2:     Choose v from {u.left, u.right} in a round-robin manner
3:     Let r_u be the root node of T_u
4:     v ← CheckTrie(u, r_u.label, r_u.hash)

5: CheckTrie(v, l_v, h_v)
6:     Let r_v ∈ u.T be the node with r_v.label = l_v
7:     if r_v ≠⊥ then
8:         if r_v.hash ≠ h_v ∧ r_v is an inner node of u.T then
9:             v ← CheckTrie(u, c_1(r_v).label, c_1(r_v).hash)
10:            v ← CheckTrie(u, c_2(r_v).label, c_2(r_v).hash)
11:    else
12:        Let c ∈ u.T with c.label = (l_v ∘ b_1 ∘ ... ∘ b_k) minimal for which l_v is a prefix
13:        if c ≠⊥ then
14:            v ← CheckAndReceive(u, c.label, c.hash, (l_v ∘ (1 − b_1)))
15:        else
16:            v ← CheckAndReceive(u, ⊥, ⊥, l_v)

17: CheckAndReceive(v, l_v, h_v, p)
18:    if l_v ≠⊥ ∧h_v ≠⊥ then
19:        CheckTrie(v, l_v, h_v)
20:    Let P be the set of all publications with the prefix p from u.T
21:    v ← Publish(P)

22: Publish(P)
23:    Insert all p ∈ P that are missing in u.T into u.T
```

CheckTrie$(v, 0, h(h(P_1) \circ h(P_2)))$ *and* *CheckTrie*$(v, 100, h(P_3)))$ *to* u*, which forces* u *to compare the hashes the nodes with labels* 0 *and* 100 *to the hashes* $h(h(P_1) \circ h(P_2))$ *and* $h(P_3)$*, respectively. Both comparisons result in the hashes being equal, which ends the chain of messages at subscriber* u*.*

Now assume that v *sends out a request* *CheckTrie*$(v, r_v.label, r_v.hash)$ *to* u *in its* *Timeout* *action. Then* u *compares* $r_v.hash$ *with* $r_u.hash$ *and spots a difference. Thus,* u *sends requests* *CheckTrie*$(u, 0, h(h(P_1) \circ h(P_2)))$ *and* *CheckTrie*$(u, 10, h(h(P_3) \circ h(P_4))))$ *to* v*. For the node with the label* 0 *this results in both hashes being equal. However, upon processing the request* *CheckTrie*$(u, 10, h(h(P_3) \circ h(P_4))))$*,* v *cannot find a node with the label* 10 *in* $v.T$*, which is why* v *sends a request* *CheckAndReceive*$(v, 100, h(P_3))$*,* $p = 101$*) to* u*. Note that the node with the label* 100 *is the node with a label of minimum length for which* 10 *is a prefix. Thus,* $p = (10 \circ (1 - 0)) = 101$*. The* *CheckAndPublish* *request forces* u *to compare the hashes of its node with the label* 100 *to the hash* $h(P_3)$*, which results in both hashes being equal. Furthermore,* u *sends all publications with labels of prefix* $p = 101$ *to* v*, which is only the publication* P_4*. In this cases* v *receives* P_4 *from* u*, resulting in both Patricia tries being equal.*

Example 9.20 shows that it makes a difference at which subscriber the initial

CheckTrie request is started. If the initial CheckTrie request is generated by u the resulting chain of messages does not spot any differences in the Patricia tries. Once v starts such a chain of messages the difference in the Patricia tries is spotted. Note that our protocol guarantees that eventually both nodes u and v initiate a CheckTrie request.

9.4.2. Analysis

We show that Algorithm 22 correctly delivers all missing publications to all subscribers in a self-stabilizing manner.

Lemma 9.21 (Publication Convergence)**.** *Consider a supervised skip ring $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$ and assume that each subscriber $u \in V$ stores a subset $P_u \subseteq P$ of publications in $u.T$ from a set P of publications such that $\bigcup_{u \in V} P_u = P$. Using Algorithm 22, the system eventually reaches a state where all subscribers u store all publications $p \in P$ in $u.T$.*

Proof. First note that in our protocol, no publish messages are deleted from the Patricia tries: i.e., once a subscriber $u \in V$ has a publication $p \in P$ stored in its Patricia trie $u.T$, it will never remove p from $u.T$. We define the potential of a pair (u, v) of subscribers by

$$\phi(u, v) = |P_{u,v} \setminus P_v|,$$

where $P_{u,v}$ is a shorthand expression for $P_u \cup P_v$. Note that $\phi(u, v) = \phi(v, u)$ does not hold in general since, intuitively speaking, $\phi(u, v)$ returns the number of publications stored in $u.T$ that are missing in $v.T$. The potential over all subscribers is then defined as

$$\Phi = \sum_{(u,v) \in E_R} \phi(u, v).$$

It is easy to see that $\Phi \geq 0$ at any point in time and $\Phi = 0 \Leftrightarrow P_u = P$ for all subscribers $u \in V$. The theorem follows if we can show that Φ is monotonically decreasing and eventually $\Phi = 0$.

We first show that Φ is monotonically decreasing. By definition, Φ increases only, if there is $(u, v) \in E_R$ for which $\phi(u, v)$ increases. This implies that there is a subscriber $w \in V$, $u \neq w \neq v$ that has sent u a set of publications $P_w \subseteq P$ via a Publish request that are not yet contained in $v.T$. Then $\phi(u, v) \leftarrow |(P_{u,v} \cup P_w) \setminus P_v|$, with $|(P_{u,v} \cup P_w) \setminus P_v| = \phi(u, v) + |P_w|$: i.e., $\phi(u, v)$ increases by $|P_w|$. But this also implies that $\phi(w, u)$ decreases by $|P_w|$, because $\phi(w, u) \leftarrow |P_{w,u} \setminus (P_u \cup P_w)|$ with $|P_{w,u} \setminus (P_u \cup P_w)| = \phi(w, u) - |P_w|$, leaving Φ at the same value as before. Thus, Φ never increases and is monotonically decreasing.

To complete the proof, we still need to show that eventually $\Phi = 0$. We do this by showing that as long as there exists $\phi(u, v) > 0$ for some edge $(u, v) \in E_R$, there is a computation after which Φ has decreased.

Let $\phi(u, v) > 0$ for two subscribers $u, v \in V$ that are connected via a ring edge $(u, v) \in E_R$. Let $p \in u.T$ be the node with the minimal label length $|p.label|$, for which it holds that all publications stored in the leaves of the subtree with root p are missing in T_v. Note that $p.label$ is a prefix for all those publications. Obviously, such

a node always exists when there is one or more publication missing in $v.T$. Assume to the contrary that we are in state s with $\phi(u, v) > 0$ and for all possible computations $\phi(u, v)$ does not decrease. We state a computation that is performed eventually in which u delivers all publications with the prefix $p.label$ to v, resulting in a decrease of $\phi(u, v)$.

W.l.o.g. consider the node $t \in u.T$ with the label length $|t.label|$ minimal, for which $t.label$ is a prefix of $p.label$ and for which there does not exist a node in $v.T$ with the label $t.label$. Such a node exists, because in case there is no inner node in $u.T$ with these properties, we can choose $t = p$. Note that we consider $p.label$ to be a prefix of itself. Consider the path $(r_u = t_1, \dots t_k = t)$ from the root node r_u of $u.T$ to t. It holds that for all nodes t_i with $i \neq k$ on this path, there exists a node $t_i' \in v.T$ with the same label as t_i, i.e., $t_i'.label = t_i.label$. Otherwise our choice for t would be wrong, since then $|t.label|$ is not minimal. As h is collision-resistant, we have for $i \in \{1, \dots, k-1\}$ that $t_i.hash \neq t_i'.hash$. We prove the following claim:

Claim 9.22. *Eventually, u sends a* CheckTrie*(u, $t.label$, $t.hash$) request to v.*

Proof. Consider the path $(r_u = t_1, \dots, t_k = t)$ from the root node r_u of $u.T$ to t. Assume that k is odd. By definition of our protocol, u will eventually send a CheckTrie(u, $r_u.label$, $r_u.hash$) request to v. As the root hashes are not equal, v sends a CheckTrie(v, $t_2.label$, $t_2.hash$) request back to u. Since we assumed that $t_i.hash \neq t_i'.hash$ for all $i \in \{1, \dots, k-1\}$, this chain continues with u sending CheckTrie(u, $t_j.label$, $t_j.hash$) requests to v, j odd, until u sends a CheckTrie(u, $t_k.label$, $t_k.hash$) request to v, which proves the claim. The case where k is even works analogously when starting at subscriber v. □

Applying Claim 9.22, we now assume that v has received a CheckTrie(u, $t.label$, $t.hash$) request from u. By our initial assumptions for t it holds that there is no node with the label $t.label$ contained in $v.T$. Thus, v searches for a node $c \in v.T$ with the label $c.label = (t.label \circ b_1 \circ \dots \circ b_k)$ of minimum length and responds to u with a CheckAndReceive(v, $c.label$, $c.hash$, p') request. Here, $p' = t.label \circ (1 - b_1)$ if c exists, otherwise $p' = t.label$.

Claim 9.23. $p' = p.label$.

Proof. For $p' = t.label$, we know that there does not exist a node with a label that has $t.label$ as a prefix. Hence, all publications with the prefix $t.label$ are missing at $v.T$, implying $t.label = p.label$, because we chose $p.label$ to be of minimal length. For $p' = t.label \circ (1 - b_1)$, we know because of the existence of $c \in v.T$ and the non-existence of a node with the label $t.label$ in $v.T$ that there is no node with the label $t.label \circ (1 - b_1)$ stored in $v.T$. Thus, all publications with the prefix $t.label \circ (1 - b_1)$ are missing in $v.T$. Since we chose $p.label$ to be of minimal length we get $t.label \circ (1 - b_1) = p.label$. □

u responds to the CheckAndReceive(v, $c.label$, $c.hash$, p') request by sending all publications in P_u to v that have the prefix p'. Since $p' = p.label$ due to Claim 9.23, $\phi(u, v)$ decreases and, since u's communication with v did not lead to an increase of $\phi(v, u)$ in this computation, Φ decreases, so the lemma follows. □

It remains to show the convergence property.

Lemma 9.24 (Publication Closure)**.** *Consider a supervised skip ring* $G = (V \cup \{s\}, E_R \cup E_S \cup E_{SUP})$ *and assume that all subscribers store the exact same Patricia trie containing all publications from a set* P*. Then no Patricia trie is modified by a subscriber as long as no subscriber generates a new publish request and no further subscriber joins the system.*

Proof. Once the system is in a legitimate state, the only type of request generated via Algorithm 22 is the periodic CheckTrie request. Assume a subscriber u has sent a request CheckTrie(u, $r_u.label$, $r_u.hash$) to v, where r_u is the root node of $u.T$. Then v compares $r_u.hash$ with the hash value $r_v.hash$ of the root node $r_v \in v.T$. Since all subscribers store the exact same Patricia trie, both hashes are equal, resulting in no further message being sent out by v as an answer to the CheckTrie request from u. □

Combing Lemmas 9.21 and 9.24 yields the main result of this section:

Theorem 9.25. *Using Algorithm 22, all publications are delivered to all subscribers in a self-stabilizing manner.*

CHAPTER 10

Conclusion and Outlook of Part II

In this chapter we conclude the second part of the thesis by summarizing our results and giving an outlook on further open problems in the area of topological self-stabilization.

Conclusion

In this thesis we presented self-stabilizing protocols for specific topologies: generalized De Bruijn graphs, quadtrees (octrees) and supervised skip rings. All of these networks are used for different purposes due to their properties. Generalized De Bruijn Graphs are useful when one wants to quickly search for participants: i.e., search requests are processed in a constant amount of hops in legitimate states. Quad- and octrees provide reliability for search requests as the routing protocol SearchQT along with the protocol BuildQT satisfy (geographic) monotonic searchability. Last but not least, we showed how to construct a self-stabilizing publish-subscribe system using the self-stabilizing supervised skip ring.

While all three topologies serve rather different purposes, they follow a bottom-up approach as a common ground. All of our protocols rely on a total order $\prec$ by which the nodes are arranged in a sorted list using BuildList. Using the list edges, further edges are established on top. We believe that such a bottom-up approach can be used in order to come up with further interesting self-stabilizing protocols for other topologies.

Outlook

Most of the self-stabilizing protocols are analyzed regarding their correctness, which means that *eventually* a legitimate state is reached when applying the protocol. Only few approaches in the literature are also analyzing the *runtime* (i.e., the worst-case time needed to reach a legitimate state) as all of our protocols rely on BuildList, for which a runtime of $\Theta(n)$ has been shown in [ORS07]. Although one has to provide a careful analysis, we suspect that $\Theta(n)$ is also the convergence time for all three of our protocols, as the time to build the remaining edges should not exceed $\mathcal{O}(n)$.

Another interesting open problem is the *congestion* generated by our protocols, meaning the maximum number of messages in a node's channel at any time. While it is easy to see that due to the round-robin approach followed in Timeout by all of our protocols, each node only generates a constant amount of new *outgoing* messages in Timeout, it is not yet clear how many *incoming* messages arrive at any node at a certain time. Ways to limit the congestion of nodes in distributed systems

are investigated by *congestion control* protocols. A protocol specifically tailored to topologically self-stabilizing systems is presented in [FGS19].

Handling joining and leaving nodes is generally not fully understood yet. While new nodes joining the system (at random existing nodes) just introduce implicit edges to the graph that are then handled by the self-stabilizing protocol, allowing nodes to safely leave the system is much more complex. As already mentioned in the related work, it has been shown that solving this problem reliably is impossible without the use of oracles [For+14]. An interesting research question would be to ask for protocols satisfying monotonic searchability in systems with leaving nodes. Here it has been shown by Scheideler et al. [SSS15] that the problem can be solved for the sorted list. It has yet to be investigated whether the problem can also be solved for other topologies that satisfy monotonic searchability. As our protocol BuildQT for the self-stabilizing quadtree relies on the sorted list and provides a quite simple mechanism to generate additional edges on top of the list, we believe that trying to extend BuildQT in order to also provide monotonic searchability under leaving nodes may be a good next step to further investigate this problem.

For self-stabilizing publish-subscribe systems, one may try to distribute the data stored by the supervisor to obtain a more decentralized solution. Here it may be helpful to introduce a network of multiple servers where each server stores a part of the original database of the supervisor, i.e., each server is responsible only for a subset of subscribers. All servers could then be arranged in a (self-stabilizing) spanning tree, for example using the protocol from Götte et al. [GSS18]. It still remains unclear whether handling Subscribe and Unsubscribe requests can be handled by the servers in a self-stabilizing manner when using the proposed structure.

Further interesting open problems such as transient behavior, locality or churn tolerance are highlighted in our survey on algorithms for self-stabilizing overlay networks [FSS20].

Bibliography

[AG96] S. V. Adve and K. Gharachorloo. “Shared Memory Consistency Models: A Tutorial”. In: *IEEE Computer* 29.12 (1996), pp. 66–76. DOI: 10.1109/2.546611.

[AGT10] S. Alaei, M. Ghodsi, and M. Toossi. “Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries”. In: *Computer Communications* 33.1 (2010), pp. 73–82. DOI: 10.1016/j.comcom.2009.08.001.

[AHS94] J. Aspnes, M. Herlihy, and N. Shavit. “Counting Networks”. In: *Journal of the ACM* 41.5 (1994), pp. 1020–1048. DOI: 10.1145/185675.185815.

[AK93] S. Aggarwal and S. Kutten. “Time Optimal Self-Stabilizing Spanning Tree Algorithms”. In: *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Bombay, India*. Vol. 761. Lecture Notes in Computer Science. Springer, 1993, pp. 400–410. DOI: 10.1007/3-540-57529-4_72.

[Alu04] S. Aluru. “Quadtrees and Octrees”. In: *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004.

[Ami+05] Y. Amir, C. Nita-Rotaru, J. R. Stanton, and G. Tsudik. “Secure Spread: An Integrated Architecture for Secure Group Communication”. In: *IEEE Transactions on Dependable and Secure Computing* 2.3 (2005), pp. 248–261. DOI: 10.1109/TDSC.2005.39.

[And+02] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. T. Morris. “Resilient overlay networks”. In: *Computer Communication Review* 32.1 (2002), p. 66. DOI: 10.1145/510726.510740.

[AS16] Y. Akhremtsev and P. Sanders. “Fast Parallel Operations on Search Trees”. In: *Proceedings of the 23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India*. IEEE Computer Society, 2016, pp. 291–300. DOI: 10.1109/HiPC.2016.042.

[AW07] J. Aspnes and Y. 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*. Ed. by E. Tovar, P. Tsigas, and H. Fouchal. Vol. 4878. Lecture Notes in Computer Science. Springer, 2007, pp. 286–300. DOI: 10.1007/978-3-540-77096-1_21.

[AW94] H. Attiya and J. L. Welch. “Sequential Consistency versus Linearizability”. In: *ACM Transactions on Computer Systems* 12.2 (1994), pp. 91–122. DOI: 10.1145/176575.176576.

[Aya90] R. Ayani. "LR-algorithm: concurrent operations on priority queues". In: *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing (SPDP), Dallas, Texas, USA*. IEEE Computer Society, 1990, pp. 22–25. DOI: 10.1109/SPDP.1990.143500.

[Bal+03] H. Balakrishnan, M. F. Kaashoek, D. R. Karger, R. T. Morris, and I. Stoica. "Looking up data in P2P systems". In: *Communications of the ACM* 46.2 (2003), pp. 43–48. DOI: 10.1145/606272.606299.

[BB98] J. Bataller and J. M. Bernabéu-Aubán. "Adaptable Distributed Shared Memory: A Formal Definition". In: *Proceedings of the 4th International Euro-Par Conference, Southampton, UK*. Vol. 1470. Lecture Notes in Computer Science. Springer, 1998, pp. 887–891. DOI: 10.1007/BFb0057944.

[BDH08] M. Ben-Or, D. Dolev, and E. N. Hoch. "Fast self-stabilizing byzantine tolerant digital clock synchronization". In: *Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing (PODC), Toronto, Canada*. ACM, 2008, pp. 385–394. DOI: 10.1145/1400751.1400802.

[Ben+13] M. Benter, M. Divband, S. Kniesburges, A. Koutsopoulos, and K. Graffi. "Ca-Re-Chord: A Churn Resistant Self-Stabilizing Chord Overlay Network". In: *2013 Conference on Networked Systems (NetSys), Stuttgart, Germany*. IEEE Computer Society, 2013, pp. 27–34. DOI: 10.1109/NetSys.2013.11.

[Ber15] A. Berns. "Avatar: A Time- and Space-Efficient Self-stabilizing Overlay Network". In: *Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Edmonton, AB, Canada*. Vol. 9212. Lecture Notes in Computer Science. Springer, 2015, pp. 233–247. DOI: 10.1007/978-3-319-21741-3_16.

[BGP13] A. Berns, S. Ghosh, and S. 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.

[BL99] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *Journal of the ACM* 46.5 (1999), pp. 720–748. DOI: 10.1145/324133.324234.

[Bru09] J. Brutlag. *Speed Matters for Google Web Search*. Tech. rep. Google, Inc, 2009.

[BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong. "Formal Aspects of Serializability in Database Concurrency Control". In: *IEEE Transactions on Software Engineering* 5.3 (1979), pp. 203–216. DOI: 10.1109/TSE.1979.234182.

[CF05] C. Cramer and T. Fuhrmann. *Self-stabilizing ring networks on connected graphs*. German. Tech. rep. 5. Universität Karlsruhe, Karlsruhe, 2005. DOI: 10.5445/IR/1000003169.

[Che52] H. Chernoff. "A measure of asymptotic efficiency for tests of a hypothesis based on the sums of observations". In: *Annals of Mathematical Statistics* 23 (1952), pp. 409–507.

[Clé+08] J. Clément, T. Hérault, S. Messika, and O. Peres. "On the Complexity of a Self-Stabilizing Spanning Tree Algorithm for Large Scale Systems". In: *14th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), Taipei, Taiwan*. IEEE Computer Society, 2008, pp. 48–55. DOI: 10.1109/PRDC.2008.36.

[CNS12] T. Clouser, M. Nesterenko, and C. Scheideler. "Tiara: A self-stabilizing deterministic skip list and skip graph". In: *Theoretical Computer Science* 428 (2012), pp. 18–35. DOI: 10.1016/j.tcs.2011.12.079.

[Coh+16] J. Cohen, J. Lefèvre, K. Maâmra, L. Pilard, and D. Sohier. "A Self-Stabilizing Algorithm for Maximal Matching in Anonymous Networks". In: *Parallel Processing Letters* 26.4 (2016), 1650016:1–1650016:17. DOI: 10.1142/S012962641650016X.

[De 46] N. G. De Bruijn. "A Combinatorial Problem". In: *Koninklijke Nederlandsche Akademie Van Wetenschappen* 49.6 (June 1946), pp. 758–764.

[DH98] M. J. Demmer and M. Herlihy. "The Arrow Distributed Directory Protocol". In: *Proceedings of the 12th International Symposium on Distributed Computing (DISC), Andros, Greece*. Vol. 1499. Lecture Notes in Computer Science. Springer, 1998, pp. 119–133. DOI: 10.1007/BFb0056478.

[Dij74] E. 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] S. Dolev and R. 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.

[Dol00] S. Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN: 0-262-04178-2.

[DS04] S. Dolev and E. Schiller. "Self-stabilizing group communication in directed networks". In: *Acta Informatica* 40.9 (2004), pp. 609–636. DOI: 10.1007/s00236-004-0143-1.

[DS07] S. Dahan and M. Sato. "Survey of Six Myths and Oversights about Distributed Hash Tables' Security". In: *27th International Conference on Distributed Computing Systems Workshops (ICDCS Workshops), Toronto, Ontario, Canada*. IEEE Computer Society, 2007, p. 26. DOI: 10.1109/ICDCSW.2007.77.

[DSW06] S. Dolev, E. Schiller, and J. L. Welch. "Random Walk for Self-Stabilizing Group Communication in Ad Hoc Networks". In: *IEEE Transactions on Mobile Computing* 5.7 (2006), pp. 893–905. DOI: 10.1109/TMC.2006.104.

[DW04] S. Dolev and J. L. Welch. "Self-stabilizing clock synchronization in the presence of Byzantine faults". In: *Journal of the ACM* 51.5 (2004), pp. 780–799. DOI: 10.1145/1017460.1017463.

[Eug+03] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. "The many faces of publish/subscribe". In: *ACM Computing Surveys* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078.

[Fab+01] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. L. M. Pereira, K. A. Ross, and D. E. Shasha. "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA*. ACM, 2001, pp. 115–126. DOI: 10.1145/375663.375677.

[FB74] R. A. Finkel and J. L. Bentley. "Quad Trees: A Data Structure for Retrieval on Composite Keys". In: *Acta Informatica* 4 (1974), pp. 1–9. DOI: 10.1007/BF00288933.

[Fel+14] P. Felber, P. Kropf, E. Schiller, and S. Serbu. "Survey on Load Balancing in Peer-to-Peer Distributed Hash Tables". In: *IEEE Communications Surveys & Tutorials* 16.1 (2014), pp. 473–492. DOI: 10.1109/SURV.2013.060313.00157.

[Fel+18] M. Feldmann, C. Kolb, C. Scheideler, and T. Strothmann. "Self-Stabilizing Supervised Publish-Subscribe Systems". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada*. IEEE Computer Society, 2018, pp. 1050–1059. DOI: 10.1109/IPDPS.2018.00114.

[FGS19] M. Feldmann, T. Götte, and C. Scheideler. "A Loosely Self-stabilizing Protocol for Randomized Congestion Control with Logarithmic Memory". In: *Proceedings of the 21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Pisa, Italy*. Vol. 11914. Lecture Notes in Computer Science. Springer, 2019, pp. 149–164. DOI: 10.1007/978-3-030-34992-9_13.

[FHS20] M. Feldmann, K. Hinnenthal, and C. Scheideler. "Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs". In: *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS), Virtual Event*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, to appear.

[FKS18] M. Feldmann, C. Kolb, and C. 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*. Vol. 11201. Lecture Notes in Computer Science. Springer, 2018, pp. 16–31. DOI: 10.1007/978-3-030-03232-6_2.

[FKS20] M. Feldmann, A. Khazraei, and C. Scheideler. "Time- and Space-Optimal Discrete Clock Synchronization in the Beeping Model". In: *Proceedings of the 32nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Virtual Event, USA*. ACM, 2020, pp. 223–233. DOI: 10.1145/3350755.3400246.

[FLS01] A. Fekete, N. A. Lynch, and A. A. Shvartsman. "Specifying and using a partitionable group communication service". In: *ACM Transactions on Computer Systems* 19.2 (2001), pp. 171–216. DOI: 10.1145/377769.377776.

[For+14] D. Foreback, A. Koutsopoulos, M. Nesterenko, C. Scheideler, and T. 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*. Vol. 8756. Lecture Notes in Computer Science. Springer, 2014, pp. 48–62. DOI: 10.1007/978-3-319-11764-5_4.

[Fre83] G. N. Frederickson. "Tradeoffs for Selection in Distributed Networks (Preliminary Version)". In: *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, Quebec, Canada*. ACM, 1983, pp. 154–160. DOI: 10.1145/800221.806718.

[FS17] M. Feldmann and C. Scheideler. "A Self-stabilizing General De Bruijn Graph". In: *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Boston, MA, USA*. Vol. 10616. Lecture Notes in Computer Science. Springer, 2017, pp. 250–264. DOI: 10.1007/978-3-319-69084-1_17.

[FS19] M. Feldmann and C. Scheideler. "Skeap & Seap: Scalable Distributed Priority Queues for Constant and Arbitrary Priorities". In: *Proceedings of the 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Phoenix, AZ, USA*. ACM, 2019, pp. 287–296. DOI: 10.1145/3323165.3323193.

[FSS18a] M. Feldmann, C. Scheideler, and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada*. IEEE Computer Society, 2018, pp. 1040–1049. DOI: 10.1109/IPDPS.2018.00113.

[FSS18b] M. Feldmann, C. Scheideler, and A. Setzer. "Skueue: A Scalable and Sequentially Consistent Distributed Queue". In: *CoRR* abs/1802.07504 (2018). arXiv: 1802.07504.

[FSS20] M. Feldmann, C. Scheideler, and S. Schmid. "Survey on Algorithms for Self-stabilizing Overlay Networks". In: *ACM Computing Surveys* 53.4 (2020), 74:1–74:24. DOI: 10.1145/3397190.

[Gal+14] D. Gall, R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. 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.

[Gao+04] J. Gao, L. J. Guibas, J. Hershberger, and L. Zhang. "Fractionally cascaded information in a sensor network". In: *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks (IPSN), Berkeley, California, USA*. ACM, 2004, pp. 311–319. DOI: 10.1145/984622.984668.

[GC07] L. Groves and R. Colvin. "Derivation of a Scalable Lock-Free Stack Algorithm". In: *Electronic Notes in Theoretical Computer Science* 187 (2007), pp. 55–74. DOI: 10.1016/j.entcs.2006.08.044.

[Gha05] S. Ghahramani. *Fundamentals of Probability with Stochastic Processes*. CRC Press, Taylor & Francis Group, Jan. 2005, pp. 249–250.

[GK93] S. Ghosh and M. H. Karaata. "A Self-Stabilizing Algorithm for Coloring Planar Graphs". In: *Distributed Computing* 7.1 (1993), pp. 55–59. DOI: 10.1007/BF02278856.

[GLR03] A. Gupta, B. Liskov, and R. Rodrigues. "One Hop Lookups for Peer-to-Peer Overlays". In: *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS), Lihue (Kauai), Hawaii, USA*. Ed. by M. B. Jones. USENIX, 2003, pp. 7–12.

[GLR04] A. Gupta, B. Liskov, and R. Rodrigues. "Efficient Routing for Peer-to-Peer Overlays". In: *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)), San Francisco, California, USA*. Ed. by R. T. Morris and S. Savage. USENIX, 2004, pp. 113–126.

[Got+98] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. "The NYU Ultracomputer - Designing a MIMD, Shared-Memory Parallel Machine". In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ACM, 1998, pp. 239–254. DOI: 10.1145/285930.285983.

[GSS18] T. Götte, C. Scheideler, and A. 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*. Vol. 11201. Lecture Notes in Computer Science. Springer, 2018, pp. 50–64. DOI: 10.1007/978-3-030-03232-6_4.

[Gum+03] P. K. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. "The impact of DHT routing geometry on resilience and proximity". In: *Proceedings of the 2003 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Karlsruhe, Germany*. ACM, 2003, pp. 381–394. DOI: 10.1145/863955.863998.

[Gup+03] I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. "Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead". In: *Peer-to-Peer Systems II, Second International Workshop (IPTPS), Berkeley, CA, USA*. Vol. 2735. Lecture Notes in Computer Science. Springer, 2003, pp. 160–169. DOI: 10.1007/978-3-540-45172-3_15.

[GVW89] J. R. Goodman, M. K. Vernon, and P. J. Woest. "Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors". In: *SIGARCH Computer Architecture News* 17.2 (1989), pp. 64–75. ISSN: 0163-5964.

[GW94] Z. Grodzki and A. Wronski. "Generalized de Bruijn graphs". In: *Journal of Information Processing and Cybernetics* 30.1 (1994), pp. 5–17.

[Hen+10] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. "Scalable Flat-Combining Based Synchronous Queues". In: *Proceedings of the 24th International Symposium on Distributed Computing (DISC), Cambridge, MA, USA*. Vol. 6343. Lecture Notes in Computer Science. Springer, 2010, pp. 79–93. DOI: 10.1007/978-3-642-15763-9_8.

[Hér+06] T. Hérault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. "Brief Announcement: Self-stabilizing Spanning Tree Algorithm for Large Scale Systems". In: *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Dallas, TX, USA*. Vol. 4280. Lecture Notes in Computer Science. Springer, 2006, pp. 574–575. DOI: 10.1007/978-3-540-49823-0_44.

[HH92] S.-C. Hsu and S.-T. Huang. "A Self-Stabilizing Algorithm for Maximal Matching". In: *Information Processing Letters* 43.2 (1992), pp. 77–81. DOI: 10.1016/0020-0190(92)90015-N.

[HMS18] B. Haeupler, J. Mohapatra, and H.-H. Su. "Optimal Gossip Algorithms for Exact and Approximate Quantile Computations". In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC), Egham, United Kingdom*. ACM, 2018, pp. 179–188. DOI: 10.1145/3212734.3212770.

[HSY10] D. Hendler, N. Shavit, and L. Yerushalmi. "A scalable lock-free stack algorithm". In: *Journal of Parallel and Distributed Computing* 70.1 (2010), pp. 1–12. DOI: 10.1016/j.jpdc.2009.08.011.

[HTW01] M. Herlihy, S. Tirthapura, and R. Wattenhofer. "Competitive concurrent distributed queuing". In: *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island, USA*. ACM, 2001, pp. 127–133. DOI: 10.1145/383962.384001.

[Hun+96] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. "An Efficient Algorithm for Concurrent Priority Queue Heaps". In: *Information Processing Letters* 60.3 (1996), pp. 151–157. DOI: 10.1016/S0020-0190(96)00148-2.

[HW90] M. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.

[II81] M. Imase and M. Itoh. "Design to Minimize Diameter on Building-Block Network". In: *IEEE Transactions on Computers* 30.6 (1981), pp. 439–442. DOI: 10.1109/TC.1981.1675809.

[Jac+12] R. Jacob, S. Ritscher, C. Scheideler, and S. 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] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. "SKIP$^+$: A Self-Stabilizing Skip Graph". In: *Journal of the ACM* 61.6 (2014), 36:1–36:26. DOI: 10.1145/2629695.

[Joh94] T. Johnson. "A Highly Concurrent Priority Queue". In: *Journal of Parallel and Distributed Computing* 22.2 (1994), pp. 367–373. DOI: 10.1006/jpdc.1994.1097.

[Kar+97] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC), El Paso, Texas, USA*. ACM, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[KF12] J. Kuo and H.-L. Fu. "On the Diameter of the Generalized Undirected De Bruijn Graphs". In: *Ars Combinatoria* 106 (2012), pp. 395–408.

[KKS12] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. "A Self-Stabilization Process for Small-World Networks". In: *26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China*. IEEE Computer Society, 2012, pp. 1261–1271. DOI: 10.1109/IPDPS.2012.115.

[KKS13] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. "CONE-DHT: A Distributed Self-Stabilizing Algorithm for a Heterogeneous Storage System". In: *Proceedings of the 27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel*. Vol. 8205. Lecture Notes in Computer Science. Springer, 2013, pp. 537–549. DOI: 10.1007/978-3-642-41527-2_37.

[KKS14] S. Kniesburges, A. Koutsopoulos, and C. 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.

[KKS15] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. "A deterministic worst-case message complexity optimal solution for resource discovery". In: *Theoretical Computer Science* 584 (2015), pp. 67–79. DOI: 10.1016/j.tcs.2014.11.027.

[KL19] P. Khanchandani and C. Lenzen. "Self-Stabilizing Byzantine Clock Synchronization with Optimal Precision". In: *Theory of Computing Systems* 63.2 (2019), pp. 261–305. DOI: 10.1007/s00224-017-9840-3.

[KLW07] F. Kuhn, T. Locher, and R. Wattenhofer. "Tight bounds for distributed selection". In: *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), San Diego, California, USA*. ACM, 2007, pp. 145–153. DOI: 10.1145/1248377.1248401.

[KNR96] D. Krizanc, L. Narayanan, and R. Raman. "Fast Deterministic Selection on Mesh-Connected Processor Arrays". In: *Algorithmica* 15.4 (1996), pp. 319–331. DOI: 10.1007/BF01961542.

[Kok+18] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 583–598. DOI: 10.1109/SP.2018.000-5.

[KP91] D. Kravets and J. K. Park. "Selection and Sorting in Totally Monotone Arrays". In: *Mathematical Systems Theory* 24.3 (1991), pp. 201–220. DOI: 10.1007/BF02090398.

[KS05] K. Kothapalli and C. Scheideler. "Supervised Peer-to-Peer Systems". In: *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN), Las Vegas, Nevada, USA*. IEEE Computer Society, 2005, pp. 188–193. DOI: 10.1109/ISPAN.2005.81.

[KS18] T. Knollmann and C. Scheideler. "A Self-stabilizing Hashed Patricia Trie". In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Tokyo, Japan*. Vol. 11201. Lecture Notes in Computer Science. Springer, 2018, pp. 1–15. DOI: 10.1007/978-3-030-03232-6_1.

[KSS17] A. Koutsopoulos, C. Scheideler, and T. 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.

[KW19] P. Khanchandani and R. Wattenhofer. "The Arvy Distributed Directory Protocol". In: *Proceedings of the 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Phoenix, AZ, USA*. ACM, 2019, pp. 225–235. DOI: 10.1145/3323165.3323181.

[KW94] B. Kröll and P. Widmayer. "Distributing a Search Tree Among a Growing Number of Processors". In: *Proceedings of the 1994 ACM International Conference on Management of Data (SIGMOD), Minneapolis, Minnesota, USA*. ACM Press, 1994, pp. 265–276. DOI: 10.1145/191839.191891.

[Lam79] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* 28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[Lam85] L. Lamport. "Solved Problems, Unsolved Problems and Non-Problems in Concurrency". In: *ACM SIGOPS: Operating Systems Review* 19.4 (1985), pp. 34–44. DOI: 10.1145/858336.858339.

[LC10] J.-C. Lin and M.-Y. Chiu. "A Fault-Containing Self-Stabilizing Algorithm for 6-Coloring Planar Graphs". In: *Journal of Information Science and Engineering* 26.1 (2010), pp. 163–181.

[LL14] C.-L. Lee and T.-J. Liu. "A Self-Stabilizing Distance-2 Edge Coloring Algorithm". In: *The Computer Journal* 57.11 (2014), pp. 1639–1648. DOI: 10.1093/comjnl/bxt072.

[LM09] A. Lakshman and P. Malik. "Cassandra: structured storage system on a P2P network". In: *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC), Calgary, Alberta, Canada*. ACM, 2009, p. 5. DOI: 10.1145/1582716.1582722.

[LSS19] L. Luo, C. Scheideler, and T. Strothmann. "MULTISKIPGRAPH: A Self-Stabilizing Overlay Network that Maintains Monotonic Searchability". In: *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil*. IEEE, 2019, pp. 845–854. DOI: 10.1109/IPDPS.2019.00093.

[Luu+16] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. "A Secure Sharding Protocol For Open Blockchains". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria*. ACM, 2016, pp. 17–30. DOI: 10.1145/2976749.2978389.

[LZ91] X. Li and F. Zhang. "On the numbers of spanning trees and Eulerian tours in generalized de Bruijn graphs". In: *Discrete Mathematics* 94.3 (1991), pp. 189–197. DOI: 10.1016/0012-365X(91)90024-V.

[Man+09] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. "A new self-stabilizing maximal matching algorithm". In: *Theoretical Computer Science* 410.14 (2009), pp. 1336–1345. DOI: 10.1016/j.tcs.2008.12.022.

[Man+93] Y. Mansour, J. K. Park, B. Schieber, and S. Sen. "Improved selection in totally monotone arrays". In: *International Journal of Computational Geometry & Applications* 3.2 (1993), pp. 115–132. DOI: 10.1142/S0218195993000087.

[Mau92] U. M. Maurer. "Asymptotically-Tight Bounds on the Number of Cycles in Generalized de Bruijn-Good Graphs". In: *Discrete Applied Mathematics* 37/38 (1992), pp. 421–436. DOI: 10.1016/0166-218X(92)90149-5.

[Mer87] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA*. Ed. by C. Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[Mor66] G. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[Mor68] D. R. Morrison. "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM* 15.4 (1968), pp. 514–534. DOI: 10.1145/321479.321481.

[Mos93] D. Mosberger. "Memory Consistency Models". In: *Operating Systems Review* 27.1 (1993), pp. 18–26. DOI: 10.1145/160551.160553.

[MS04] M. Moir and N. Shavit. "Concurrent Data Structures". In: *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. DOI: 10.1201/9781420035179.

[MS96] M. M. Michael and M. L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC), Philadelphia, Pennsylvania, USA*. ACM, 1996, pp. 267–275. DOI: 10.1145/248052.248106.

[MS98] M. M. Michael and M. L. Scott. "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors". In: *Journal of Parallel and Distributed Computing* 51.1 (1998), pp. 1–26. DOI: 10.1006/jpdc.1998.1446.

[Müh+05] G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, A. Ulbrich, and L. Fiege. "Self-stabilizing Publish/Subscribe Systems: Algorithms and Evaluation". In: *Proceedings of the 11th European Conference on Parallel Processing (EUROPAR), Lisbon, Portugal*. Vol. 3648. Lecture Notes in Computer Science. Springer, 2005, pp. 664–674. DOI: 10.1007/11549468_73.

[NNS13] R. M. Nor, M. Nesterenko, and C. 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.

[NW07] M. Naor and U. Wieder. "Novel architectures for P2P applications: The continuous-discrete approach". In: *ACM Transactions on Algorithms* 3.3 (2007), p. 34. DOI: 10.1145/1273340.1273350.

[OL13] J. Ou and J. Li. "On m-restricted edge connectivity of undirected generalized De Bruijn graphs". In: *International Journal of Computer Mathematics* 90.11 (2013), pp. 2259–2264. DOI: 10.1080/00207160.2013.778984.

[ORS07] M. Onus, A. W. Richa, and C. Scheideler. "Linearization: Locally Self-Stabilizing Sorting in Graphs". In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, Louisiana, USA*. SIAM, 2007. DOI: 10.1137/1.9781611972870.10.

[PRU03] G. Pandurangan, P. Raghavan, and E. Upfal. "Building low-diameter peer-to-peer networks". In: *IEEE Journal of Selected Areas in Communications* 21.6 (2003), pp. 995–1002. DOI: 10.1109/JSAC.2003.814666.

[RD01] A. I. T. Rowstron and P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*. Vol. 2218. Lecture Notes in Computer Science. Springer, 2001, pp. 329–350. DOI: 10.1007/3-540-45518-3_18.

[RLS02] R. Rodrigues, B. Liskov, and L. Shrira. "The design of a robust peer-to-peer system". In: *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France*. ACM, 2002, pp. 117–124. DOI: 10.1145/1133373.1133396.

[RPK80] S. M. Reddy, D. K. Pradhan, and J. G. Kuhl. *Directed graphs with minimum diameter and maximum connectivity*. Tech. rep. School of Engineering, Oakland University, 1980.

[RS04a] C. Riley and C. Scheideler. "A Distributed Hash Table for Computational Grids". In: *Proceedings of the 2004 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004. DOI: 10.1109/IPDPS.2004.1302971.

[RS04b] C. Riley and C. Scheideler. "Guaranteed broadcasting using SPON: supervised P2P overlay network". In: *International Zurich Seminar on Communications, 2004*. 2004, pp. 172–175.

[RS97] S. Rajasekaran and S. Sahni. "Sorting, Selection, and Routing on the Array with Reconfigurable Optical Buses". In: *IEEE Transactions on Parallel and Distributed Systems* 8.11 (1997), pp. 1123–1132. DOI: 10.1109/71.642947.

[RSS11] A. W. Richa, C. Scheideler, and P. Stevens. "Self-Stabilizing De Bruijn Networks". In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Grenoble, France*. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 416–430. DOI: 10.1007/978-3-642-24550-3_31.

[RSS86] D. Rotem, N. Santoro, and J. B. Sidney. "Shout echo selection in distributed files". In: *Networks* 16.1 (1986), pp. 77–86. DOI: 10.1002/net.3230160108.

[Sam89] H. Samet. "Hierarchical Spatial Data Structures". In: *Proceedings of the 1st Symposium on Design and Implementation of Large Spatial Databases (SSD), Santa Barbara, California, USA*. Vol. 409. Lecture Notes in Computer Science. Springer, 1989, pp. 193–212. DOI: 10.1007/3-540-52208-5_28.

[SB15] G. Sharma and C. Busch. "Distributed Queuing in Dynamic Networks". In: *Parallel Processing Letters* 25.2 (2015), 1550005:1–1550005:17. DOI: 10.1142/S012962641550005X.

[SL00] N. Shavit and I. Lotan. "Skiplist-Based Concurrent Priority Queues". In: *Proceedings of the 2000 IEEE International Parallel & Distributed Processing Symposium (IPDPS), Cancun, Mexico*. IEEE Computer Society, 2000, pp. 263–268. DOI: 10.1109/IPDPS.2000.845994.

[SM02] E. Sit and R. T. Morris. "Security Considerations for Peer-to-Peer Distributed Hash Tables". In: *Peer-to-Peer Systems, First International Workshop (IPTPS), Cambridge, MA, USA*. Vol. 2429. Lecture Notes in Computer Science. Springer, 2002, pp. 261–269. DOI: 10.1007/3-540-45748-8_25.

[SN04] R. C. Steinke and G. J. Nutt. "A unified theory of shared memory consistency". In: *Journal of the ACM* 51.5 (2004), pp. 800–849. DOI: 10.1145/1017460.1017464.

[SR05] A. Shaker and D. 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*. IEEE Computer Society, 2005, pp. 39–46. DOI: 10.1109/P2P.2005.34.

[SS05] C. Schindelhauer and G. Schomaker. "Weighted Distributed Hash Tables". In: *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Las Vegas, Nevada, USA*. ACM, 2005, pp. 218–227. DOI: 10.1145/1073970.1074008.

[SS09] C. Scheideler and S. Schmid. "A Distributed and Oblivious Heap". In: *Automata, Languages and Programming, 36th Internatilonal Colloquium (ICALP), Rhodes, Greece*. Vol. 5556. Lecture Notes in Computer Science. Springer, 2009, pp. 571–582. DOI: 10.1007/978-3-642-02930-1_47.

[SS93] S. Sur and P. K. Srimani. "A self-stabilizing algorithm for coloring bipartite graphs". In: *Information Sciences* 69.3 (1993), pp. 219–227. DOI: 10.1016/0020-0255(93)90121-2.

[SSO94] Y. Shibata, M. Shirahata, and S. Osawa. "Counting Closed Walks in Generalized de Bruijn Graphs". In: *Information Processing Letters* 49.3 (1994), pp. 135–138. DOI: 10.1016/0020-0190(94)90090-6.

[SSS15] C. Scheideler, A. Setzer, and T. 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*. Vol. 46. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, 24:1–24:17. DOI: 10.4230/LIPIcs.OPODIS.2015.24.

[SSS16] C. Scheideler, A. Setzer, and T. 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*. Vol. 9888. Lecture Notes in Computer Science. Springer, 2016, pp. 71–84. DOI: 10.1007/978-3-662-53426-7_6.

[SSW02] K. Schlude, E. Soisalon-Soininen, and P. Widmayer. "Distributed Highly Available Search Trees". In: *Proceedings of the 9th International Colloquium on Structural Information and Communication Complexity (SIROCCO), Andros, Greece*. Vol. 13. Proceedings in Informatics. Carleton Scientific, 2002, pp. 259–274.

[SSW03] K. Schlude, E. Soisalon-Soininen, and P. Widmayer. "Distributed Search Trees: Fault Tolerance in an Asynchronous Environment". In: *Theory of Computing Systems* 36.6 (2003), pp. 611–629. DOI: 10.1007/s00224-003-1121-7.

[ST05] H. Sundell and P. Tsigas. "Fast and lock-free concurrent priority queues for multi-thread systems". In: *Journal of Parallel and Distributed Computing* 65.5 (2005), pp. 609–627. DOI: 10.1016/j.jpdc.2004.12.005.

[ST16a] N. Shavit and G. Taubenfeld. "The computability of relaxed data structures: queues and stacks as examples". In: *Distributed Computing* 29.5 (2016), pp. 395–407. DOI: 10.1007/s00446-016-0272-0.

[ST16b] G. Siegemund and V. Turau. "*PSVR* - Self-stabilizing Publish/Subscribe Communication for Ad-Hoc Networks (Short Paper)". In: *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Lyon, France*. Vol. 10083. Lecture Notes in Computer Science. 2016, pp. 346–351. DOI: 10.1007/978-3-319-49259-9_27.

[ST18] G. Siegemund and V. Turau. "A Self-Stabilizing Publish/Subscribe Middleware for IoT Applications". In: *ACM Transactions on Cyber-Physical Systems* 2.2 (2018), 12:1–12:26. DOI: 10.1145/3185509.

[ST97] N. Shavit and D. Touitou. "Elimination Trees and the Construction of Pools and Stacks". In: *Theory of Computing Systems* 30.6 (1997), pp. 645–670. DOI: 10.1007/s002240000072.

[STM15] G. Siegemund, V. Turau, and K. Maamra. "A self-stabilizing publish/subscribe middleware for wireless sensor networks". In: *2015 International Conference and Workshops on Networked Systems, (NetSys), Cottbus, Germany*. IEEE Computer Society, 2015, pp. 1–8. DOI: 10.1109/NetSys.2015.7089067.

[Sto+01] I. Stoica, R. T. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *Proceedings of the 2001 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), San Diego, CA, USA*. ACM, 2001, pp. 149–160. DOI: 10.1145/383059.383071.

[SZ00] N. Shavit and A. Zemach. "Combining Funnels: A Dynamic Approach to Software Combining". In: *Journal of Parallel and Distributed Computing* 60.11 (2000), pp. 1355–1387. DOI: 10.1006/jpdc.2000.1621.

[SZ99] N. Shavit and A. Zemach. “Scalable Concurrent Priority Queue Algorithms”. In: *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC), Atlanta, Georgia, USA*. ACM, 1999, pp. 113–122. DOI: 10.1145/301308.301339.

[TH06] S. Tirthapura and M. Herlihy. “Self-Stabilizing Distributed Queuing”. In: *IEEE Transactions on Parallel and Distributed Systems* 17.7 (2006), pp. 646–655. DOI: 10.1109/TPDS.2006.94.

[THS07] E. Tanin, A. Harwood, and H. Samet. “Using a distributed quadtree index in peer-to-peer networks”. In: *The VLDB Journal* 16.2 (2007), pp. 165–178. DOI: 10.1007/s00778-005-0001-y.

[YT10] Y. Yamauchi and S. Tixeuil. “Monotonic Stabilization”. In: *Proceedings of the 14th International Conference on Principles of Distributed Systems (OPODIS), Tozeur, Tunisia*. Vol. 6490. Lecture Notes in Computer Science. Springer, 2010, pp. 475–490. DOI: 10.1007/978-3-642-17653-1_34.

[Zha+04] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. “Tapestry: a resilient global-scale overlay for service deployment”. In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. DOI: 10.1109/JSAC.2003.818784.

List of Algorithms and Figures

List of Algorithms

List of Figures