Distributed Data Structures and the Power of Topological Self-Stabilization

Dissertation by
Sebastian Kniesburges
University of Paderborn

Reviewers:

- Prof. Dr. Christian Scheideler, University of Paderborn
- Prof. Dr. Friedhelm Meyer auf der Heide

To Miriam my little sunshine on cloudy days.

Acknowledgments

Working on this thesis was only possible by the help of several people to whom I like to show how grateful I am. One of them is my supervisor Christian Scheideler who always gave helpful advice and provided new research directions in times research seemed to reach a dead end. I further like to thank my colleague and office mate Andreas Koutsopoulos with whom I shared the joy and struggle of finding proofs for our protocols and theorems, writing papers and waiting for their submission and visiting conferences around the world. Of course many thanks go to all my colleagues during my time in our research group for providing fruitful discussions during the lunch time whether it was about football, politics, economics or any other topic. I will also always remember our Kicker sessions after lunch. I also thank all other colleagues. I enjoyed spending my time in the cake seminar on every Friday and playing football in our weekly Theory Soccer Sessions. Many thanks also go to my family. You are always there when I need you. Most of all, I thank my little daughter, whose smiles can brighten the darkest days. Thank You Miriam, although playing with you did not help to finish this thesis.

Zusammenfassung

In dieser Arbeit betrachten wir Probleme im Bereich verteilter Systeme und lokaler Algorithmen. Wir betrachten verteilte Systeme die gegeben sind durch bestimmte Topologien miteinander vernetzen Knoten und stellen die Frage, ob solche Topologien wiederhergestellt werden können wenn das Netzwerk durch den Ausfall von Knoten oder Kanten oder dem Hinzukommen neuer Knoten oder Kanten verändert wird. Dabei sollen lokale verteilte Algorithmen entwickelt werden, d.h. die Algorithmen werden auf jedem Knoten ausgeführt und nutzen nur Informationen die lokal beim Knoten gespeichert wird wie z.B. seine benachbarten Knoten, die das Netzwerk von einer beliebigen schwach zusammenhängenden Starttopologie in eine gewünschte Zieltopologie überführen.

Diese Eigenschaft eines Algorithmus nennen wir topologische Selbststabilisierung. Motiviert wird diese Betrachtung durch die zunehmende Nutzung von Peer-to-Peer (P2P) Systemen und von Cloud Dienstleistern, also Szenarien in denen das verteilte System aus Ressourcen besteht, für die Ausfälle und Verbindungen nicht mehr kontrolliert werden können.

Zur Analyse von topologisch selbststabilisierenden Algorithmen oder Protokollen führen wir geeignete Modelle ein. Wir präsentieren dann für einige bestimme Topologien, mit welchen topologisch selbststabilisierenden Protokollen diese erreicht werden können. Wir betrachten dabei als einführendes Beispiel eine sortierte Liste von Knoten und fahren dann mit komplexeren Topologien wie einem bestimmtem Small-World Netzwerk und einem komplettem Graphen fort.

Als nächstes wenden wir die Idee von topologisch selbststabilisierenden Protokollen auf das bekannte Konzept von verteilten Hashtabellen an. Dabei zeigen wir, dass eine solche Lösung für bereits existierende verteilte Hashtabellen möglich ist und entwickeln dann eine weitere verteilte Hashtabelle, die heterogene Kapazitäten unterstützt, und ein dazugehöriges Protokoll.

Zum Schluss verlassen wir den Bereich topologisch selbststabilisierender Protokolle und betrachten stattdessen, wie verteilte Hashtabellen erweitert werden können, sodass nicht nur exakte Suchanfragen unterstützt werden sondern auch Suchanfrage nach dem ähnlichstem Schlüssel. Dabei betrachten wir zum einen Ähnlichkeit in dem Sinne, dass der Antwortschlüssel den längsten gemeinsamen Präfix mit dem Suchschlüssel besitzt, und zum anderen Ähnlichkeit in dem Sinne, dass der Antwortschlüssel der Vorgänger des Suchschlüssels in einer gegebenen Ordnung ist.

Abstract

This thesis considers problems located in the fields of distributed systems and local algorithms. In particular we consider such systems given by specific topologies of interconnected nodes and want to examine whether these topologies can be rebuilt in case the network is (massively) changed by failing or joining nodes or edges. For this case we search for local distributed algorithms, i.e. the algorithms are executed on every single node and only use local information stored at the nodes like their neighborhood of nodes. By executing these algorithms we will show that the desired goal topologies can be reached from any weakly connected start topology.

We call this property of an algorithm topological self-stabilization and motivate it by the increasing usage of peer-to-peer (P2P) systems and of cloud computing. In both cases the user or owner of the data and executed algorithms cannot control the resources and their connectivity.

In order to analyze topological self-stabilizing algorithms or protocols we introduce suited models. For some specific topologies we then present and analyze topological self-stabilizing protocols. We consider topologies like a sorted list of nodes, which we use as a simple introductory example. We then proceed with more complex topologies like a specific small-world network and a clique.

We then show that the concept of topological self-stabilization can be used for distributed hash tables. In particular we show that for existing distributed hash tables a topological self-stabilizing protocol can be found. We also construct a new overlay network, that builds a distributed hash table that supports heterogeneous capacities, and a corresponding topological self-stabilizing protocol.

At last we leave the concept of topological self-stabilization behind and instead show how to extend the usage of distributed hash tables, in order to answer more than only exact queries. We present data structures that can be built on top of distributed hash tables and support longest prefix queries and predecessor or range queries.

Contents

CHAPTER 1

Introduction

Imagine you're the next Mark Zuckerberg. You've got the idea for the next big thing. As you are an IT expert you are aware of all the new trends and technologies. As you also lack of money to invest in infrastructure like several servers with high bandwidth, you decide to follow the current trend of distributed computing and instead of buying the infrastructure you use the storage, computational power and bandwidth provided by the so called cloud providers or by other users in a peer-to-peer network. But with this decision new problems arise. If you use infrastructure provided by others you no longer control this infrastructure, so the challenge is how to react on failures, on down times of the machines you rented. Although cloud providers give guarantees on the availability of the resources you rented, e.g. the Service Level Agreement of Amazon EC2 claims an availability of 99.95% [62], unavailability can still be a problem especially if you have several resources provided by different cloud providers that should be interconnected.

Let's see if you can imagine something more. Let's assume that if your application is truly successful it will run on maybe thousands of virtual machines provided by different cloud providers (e.g. Google, Amazon, Microsoft azure) [61, 28, 15] that should be interconnected in a specific way. In our example we assume that all virtual machines are numbered, i.e. there exists a machine 1, a machine 2,$\cdots$. Let's further assume that machine 1 should be connected to machines 2-4, machine 2 to machines 3-5, machine 3 to machines 4-6 and so on. A question that arises is how this connections can be maintained in case some machines are unavailable. Let's assume machine 6 is unavailable, then machine 7 assumes the role of machine 6, machine 8 assumes the role of machine 7 and so on. Thus some connections have to be changed in case machines become unavailable, e.g. machine 3 now has to be connected to machine 7 instead of the unavailable machine 6.

As we assumed that our application is running on several thousands of machines unavailability of some machines becomes more likely and the connections can not be maintained manually. In fact we want our system to be able to recover from any kind of degenerated state it can get into by failing or unavailable machines or communication links. In this thesis we show some solutions to such problems in the field of distributed systems. See Figure 1.1

In recent years a new quality of dynamics and uncertainty has been introduced to the concept of distributed systems. Assumptions like that the number of nodes or faulty nodes in the system, bounds of these or bounds on the network diameter are known before and by every node and can be used as parameters in the distributed algorithm no longer hold if one considers distributed systems which are peer-to-peer based or built using resources in the cloud [4, 57].

In peer-to-peer systems peers are allowed to enter or leave the system dynamically. It follows that

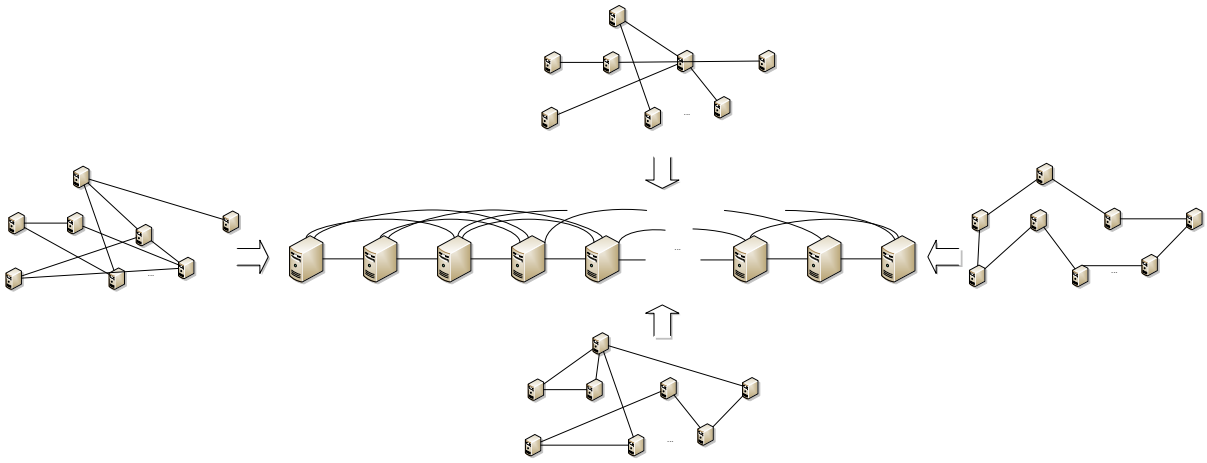


Figure 1.1: Recovery of the connections of the set of virtual machines

no node can be aware of the total number of nodes whether faulty or not currently in the system. Thus such an assumption is questioned from a practical point of view. By the leaving and (re)joining of nodes also communication paths in the system alter or communication by some pair of nodes is even no longer possible. So also assumptions on the network diameter or communication paths in general are not possible in these settings.

A similar form of dynamics is introduced by the cloud computing paradigm. The cloud computing paradigm states the need of elasticity and scalability in order to manage variable and potentially huge workloads. This is achieved by scaling the storage or computational power of a single resource on the one hand but also by adding further resources like virtual machines dynamically. So in such a setting also the number of machines cannot be predicted in advance or is known by a single machine.

These new uncertainties are added to the several uncertainties already existing in static distributed systems like the lack of temporal knowledge of the nodes or unreliable communication which are typically modeled by asynchronous systems with different message passing models.

In this thesis we will present some answers on how to handle the challenges resulting from the described dynamics. We will show that even if nodes join or leave the system or if communication between nodes fail, i.e. if edges fail, the system can recover such that assumptions on communication paths or the network diameter depending on the number of nodes are possible again. In fact we provide distributed algorithms that can recover specific topologies and these topologies then provide bounds on the network diameter or define communication paths. As described above such solutions can be helpful for peer-to-peer systems or systems based on the cloud computing paradigm.

The presented techniques can be reused for peer-to-peer systems to recover the topology interconnecting the peers. Most peer-to-peer systems claim to be self-organizing in the sense that the joining and leaving of nodes can be handled if the rest of the system works properly, i.e. if all other nodes form the correct topology and that no edge failures occur. With these restrictions it is unlikely that a worst-case scenario happens and the topology interconnecting the peers degenerates totally [75, 69, 70, 50]. With our solutions these restrictions are no longer necessary as with our distributed algorithms the topology can be recovered from any degenerated form as long as all peers stay connected.

In the cloud computing scenario our techniques can be used to interconnect different resources like

virtual machines that are provided by different cloud providers. So our solutions can be applied to the case of cloud federation as our distributed algorithms maintain topologies automatically even if resources are added or are taken out of the system dynamically. If a user needs further resources and rents further virtual machines by some cloud provider the user does not have to connect these machines manually to the machines already in his distributed system instead he only has to run one of the distributed algorithms presented in this thesis on the virtual machines to maintain a specific topology on the virtual machines.

1.1 Thesis Overview

This thesis consists of four parts. In the chapter 2 we introduce the network and computational model we will use in the following two parts. We further define topological self-stabilization.

In chapter 3 we develop topological self-stabilizing protocols for three different overlay networks. We consider the sorted list, a specific small-world network and a clique as the overlay networks and show the correctness of the corresponding protocols and analyze their stabilization time, stabilization work and maintenance work.

In chapter 4 we apply our techniques of self-stabilizing protocols to distributed hash tables. We present a Chord like overlay network Re-Chord that maintains all the features of Chord and a corresponding topological self-stabilizing protocol. We then develop a new overlay network that supports heterogeneous capacities of the nodes and also a topological self-stabilizing protocol.

In chapter 5 we do not develop new overlay networks or topological self-stabilizing protocols but show how to extend existing solutions to support more than only exact queries. In particular we present techniques to allow prefix queries and predecessor queries on distributed hash tables by providing two data structures, the hashed Patricia trie and the hashed predecessor Patricia trie that can be built on top of distributed hash tables.

CHAPTER 2

Models

In this chapter we introduce our models for topological self-stabilization. In this thesis two models for topological self-stabilization are presented: an asynchronous model *ATSS* (asynchronous topological self-stabilization), that will be used to give strong results on the correctness of our protocols, as our protocols even work in an asynchronous setting, and a synchronous model *STSS* (synchronous topological self-stabilization), that we use to provide bounds on the stabilization time and other complexities of our protocols.

2.1 Topological Self-Stabilization

In this section we describe the general model for topological self-stabilization by giving a description of the used network and computational models and then define what we mean by topological self-stabilization.

2.1.1 Network model

We assume a message passing model which is related to the model presented in [59] by Nor et al. To model the structure of the networks we use a graph theoretic approach. The overlay network consists of a static set V of n nodes and a set of edges E, which will be described in detail later on. We assume *fixed identifiers* (ids) for each node. These identifiers are *immutable* in the computation, we only allow identifiers to be compared, stored and sent. In our model the identifiers are used as addresses, such that by knowing the identifier of one node another node can send messages to this node. On the identifiers a unique order can be established. We further assume that there are no false identifiers in the network. Although this is a strong assumption it can be justified. If we assume that there are false identifiers in the system we need some mechanism to detect them before a message is sent to the false identifier. Thus false identifiers don't affect the protocol used to build the topologies, as still messages will only be sent between nodes with correct identifiers. As we focus on the self-stabilizing protocols to build certain topologies, we can simply assume that false identifiers can be detected or in other words no false identifiers are existing.

The communication between nodes is realized by passing messages through channels. A node v can send a message to u through the channel $Ch_{v,u}$. We assume that the capacity of a channel $Ch_{v,u}$ is unbounded and no messages are lost. We denote the channel Ch_u as the union of all channels $Ch_{v,u}$.

In order to describe the edge set E of the graph modeling our network we firstly define the computational states. We distinguish between the *node state* of a node u, that is given by the set of identifiers stored in

the internal variables of u, i.e. the nodes u can communicate with, and the *channel state* of a channel Ch_u, that is given by all identifiers contained in messages in a channel Ch_u. The *program state* is then defined by the node states of all nodes and the channel states of all channels, i.e. the assignment of values to every variable of each node and messages to every channel. We call the combination of the node states of all nodes the *node state of the system* and the combination of the channel states of all channels the *channel state of the system*.

We model the network by a directed graph $G = (V, E)$ called *communication graph*. The set of edges E describes the possible communication pairs. E consists of two subsets: the *explicit edges* $E_e = \{(u, v) : v \text{ is in u's node state}\}$ and the *implicit edges* $E_i = \{(u, v) : v \text{ is in the channel state of} Ch_u\}$. Then $E = E_e \cup E_i$. Moreover we define $G_e = (V, E_e)$ as the *explicit communication graph*. So we can define a graph corresponding to every program state. We call G^t and G_e^t the *communication graph* and *explicit communication graph* for an program state at time t.

2.1.2 Computational Model

Each node performs some predefined protocol P. A protocol P consists of a set of actions. An action has the form $< guard > \rightarrow < command >$. The *guard* predicate can be true or false, the guard predicate only depends on local information of a node, i.e. its node state and a received message. *command* is a sequence of statements that may perform computations or send messages to other nodes. Also all computations only depend on the local information of the node performing the computations. Messages can only be sent to nodes of which the identifiers are in the node state of the sending node or in the received message. We introduce one special guard predicate τ called the *timer predicate*, which is periodically true; i.e. according to an internal clock τ becomes true after a number of clock cycles and is false the other times. We use τ to allow the nodes to perform periodical actions. A second predicate is true if a message is received by a node. An action is enabled in some state if its guard is true and disabled otherwise.

A *computation* is a sequence of program states such that for each program state s_i the next program state s_{i+1} is reached by executing an enabled action in s_i. By this definition, actions can not overlap and are executed atomically giving a sequential order of the executions of actions. Thus by executing an enabled action we reach the graph G^{i+1} from the graph G^i.

2.1.3 Topological Self-stabilization

We now formally describe the problem of topological self-stabilization. In topological self-stabilization the goal is to state a protocol *P* that *solves* an overlay problem *OP* starting from a topology in the set *IT*. IT is the set of possible initial topologies. A protocol is *unconditionally* self-stabilizing if *IT* contains every possible state. Analogously a protocol is *conditionally* self-stabilizing if *IT* contains only states that fulfill some conditions. For topological self-stabilization we assume that *IT* contains any state as long as $G^{IT} = \left(V, E^{IT}\right)$ is weakly connected, i.e. the combined knowledge of all nodes and channels in this state covers the whole network. Furthermore there are no identifiers that don't belong to existing nodes in the network.

The set of target topologies defined in *OP* is given by $OP = \left\{G_e^{OP} = \left(V, E_e^{OP}\right)\right\}$, i.e. the goal topologies of the overlay problem are only defined on explicit edges and E_i^{OP} can be an arbitrary (even empty) set of edges. We define the set of target topologies by the explicit edges only, as these are permanent edges, i.e. ids stored in internal variables. So as soon as the topology formed by explicit edges is in the set of target topologies, this topology fulfills all properties of the target topologies e.g. short

diameter, short routing length etc. despite any still existing implicit edges. We also call the program states in *OP legal states*. We say a protocol *P* that solves a problem *OP* is topologically self-stabilizing if for *P* *convergence* and *closure* can be shown. *Convergence* means that for *P* started with any state in *IT* every computation eventually leads to a legal state in *OP*. *Closure* means that *P* started in a legal state in *OP* maintains a legal state, i.e. only legal states are reachable. An illustration of the convergence and closure property of a protocol *P* is given in Figures 2.1 and 2.2.

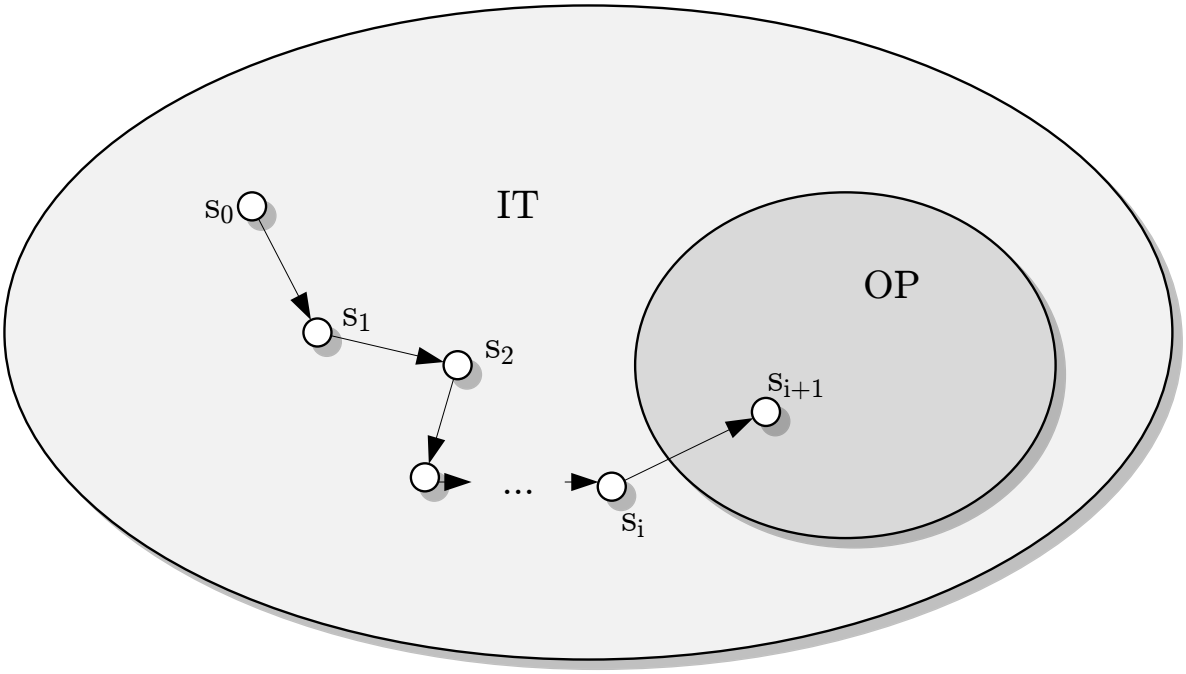


Figure 2.1: Convergence for a protocol *P*

For a protocol *P* we assume that there are no oracles available for the computation. In particular we assume that there is no *connection oracle*, that can connect disconnected parts of the network, no *identifier detector*, that can decide whether an identifier belongs to an existing node or not, and no *legal state detector*, that can decide based on global knowledge whether the system is in a legal state or not. With these assumptions our model complies with the *compare-store-send* program model in [59] in which protocols do not manipulate the internals of the nodes' identifiers.

2.2 ATSS vs. STSS

In the ATSS model we assume an asynchronous message passing, i.e. for a message in a channel Ch_u it is unknown when the message is delivered. For the channel we assume *eventual delivery* meaning that if there is a state in the computation where there is a message m in the channel Ch_u there also is a later state where the message m is not in the channel, but was received by the process. Note that this does not imply any order between the messages from sending nodes. Considering the computational model we add the assumption of *weak fairness* to our model. By weak fairness we mean that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. We

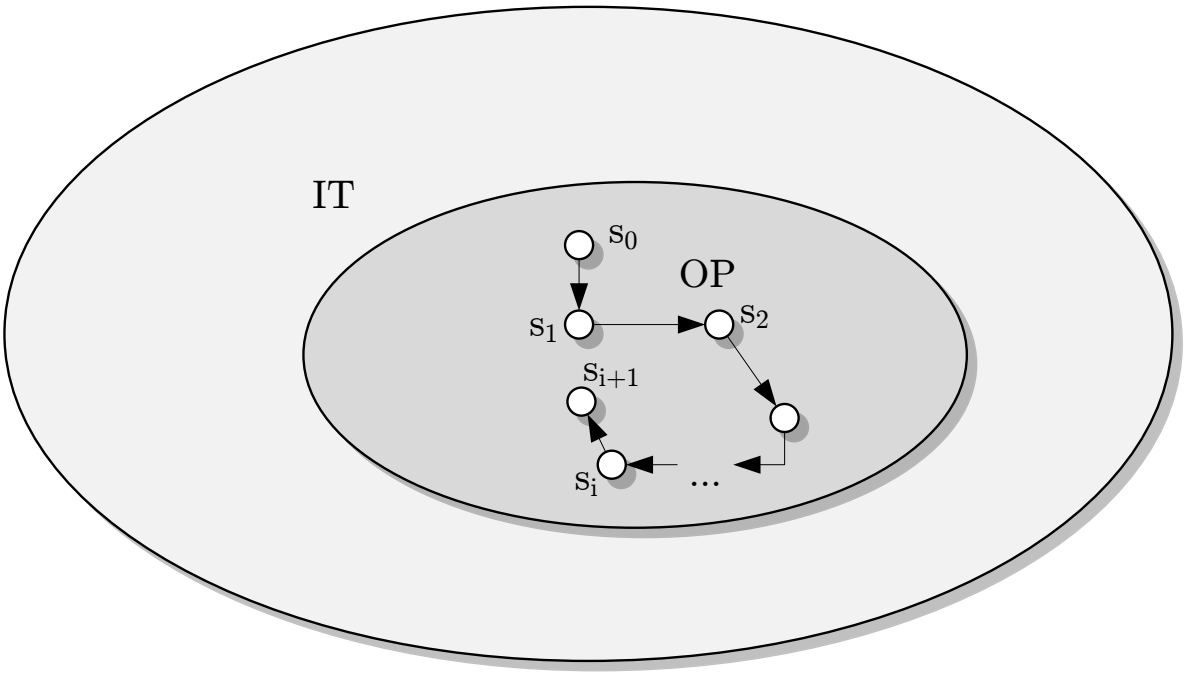


Figure 2.2: Closure for a protocol *P*

further assume arbitrary asynchronicity concerning the internal clock of each node. Considering this strong asynchronicity in the ATSS model we use this model in the following to show the correctness of our protocols. By proving their correctness in the ATSS model we show that our protocols can be implemented in a real-world application as also the internet is asynchronous and based on point-to-point communication. For the ATSS model we don't define asynchronous rounds to give any bounds on the complexity of our protocols in an asynchronous setting. For this purpose we will introduce a second synchronous model STSS.

In the STSS model we assume a synchronous message passing model, i.e. in the STSS model the protocol is executed in synchronous rounds. For the passing of messages this means that messages sent in one round are received immediately before the next round. Thus a node executes the action depending on the received message one round after this messages was set. For the computational model this means that we can drop the assumption of *weak fairness* and instead assume that each action that is enabled in one round is also executed in the same round. For the STSS model we define that the next program state s_{t+1} is reached from s_t by executing all enabled actions in round t. Accordingly the set $E^t = E_i^t \cup E_e^t$ are defined. Thus by executing all enabled actions in one round we reach the graph G^{t+1} from the graph G^t. In this thesis we use the STSS model to give complexity bounds for our protocols in a synchronous setting to be able to talk about the efficiency of our protocols. We define the *stabilization time* to be the worst-case number of rounds until, according to a protocol P, a state in OP is reached from any initial state in IT. We further define two different kinds of work complexities.

We define the *stabilization work* to be the worst-case number of messages each node sends or receives until according to a protocol P a state in OP is reached from any initial state in IT. For the stabilization work we do not take into account the messages already in the initial state. As the channels capacities

ATSS	STSS
asynchronous message passing	*synchronous message passing*
eventual delivery of messages	*messages are received in the next round*
weak fairness	*enabled action are executed in each round*
correctness only	*complexity results*

Table 2.1: Properties of the two models for topological self-stabilization

are unbounded, there can be an arbitrary number of messages initially in the system, and thus also the work in terms of received messages is unbounded. Therefore we only consider the messages sent and received during the stabilization process, i.e. messages sent in the first round or later. For our analysis we assume that each message contains a single identifier. If in our protocols messages are sent or received that contain i identifiers then we simply count it as sending or receiving i messages.

The *maintenance work* is defined as the worst-case number of messages each node sends or receives in one round in a state in OP, where we do not take into account the messages resulting from the stabilization, i.e. we assume that there remain no messages in the Channel of each node after the stabilization. We call such a stable state without remaining implicit edges OP^*.

Furthermore we give bounds on the number of rounds and worst-case work for a node in the network to recover the topology from single structural changes. We consider only single *join* and *leave* events in a legal state. This means we want to answer the following question: in case the topology is already stable how long does it take to recover if one single node joins or leaves the network? In case of a leaving node we assume that the leaving of the node does not disconnect the network. Otherwise we say that a node is not allowed to leave. We give a short overview of the two described models in table 2.1.

CHAPTER 3

Self-Stabilizing Overlay Networks

3.1 Introduction

In this chapter we describe topological self-stabilizing algorithms that form overlay networks. We only consider the protocols forming certain topologies and not the application of such topologies as distributed data structures storing information, e.g. distributed hash tables. We start by considering a simple topology to demonstrate how a topological self-stabilizing protocol looks like and analyze it according to our two models. In fact we give a protocol for a self-stabilizing sorted list in section 3.2. We start with the sorted list as it is not only a simple problem but the used protocol will also be reused to develop protocols for more complex topologies. We present a protocol P^{LIST} and an improved protocol $P^{LISTsync}$ for this overlay problem and show that it is indeed topologically self-stabilizing in the ATSS and STSS model. We further show a stabilization time of $\Theta(n)$, a stabilization work of $\Theta(n^2)$ (resp. $\Theta(n)$ for $P^{LISTsync}$) and a maintenance work of $\mathcal{O}(1)$.

More complex topologies for which we present self-stabilizing protocols are small-world networks and the complete graph. We consider small-world networks in section 3.3 as they share properties with many real-world networks. For this overlay problem we again develop two protocols $P^{SMALL-WORLD}$ and $P^{SMALL-WORLDsync}$ that are self-stabilizing in the ATSS and STSS model. We additionally show a stabilization time of $\mathcal{O}(n+\delta)$ and a stabilization work of $\mathcal{O}(n^2)$, and a maintenance work of $\mathcal{O}(n)$ for the protocol $P^{SMALL-WORLDsync}$ if δ is the converging time of a specific random process.

At last we consider the problem of forming a complete graph in section 3.4. The complete graph is an interesting overlay problem as it also is a way to discover all nodes in the network. This problem is also known as resource discovery. If a node is aware of every resource in the network it is also able to communicate with every resource and, according to our model, is therefore connected to every other resource, i.e. all nodes form a complete graph. We give a topological self-stabilizing protocol P^{CLIQUE} for this overlay problem of forming a complete graph or a clique that is correct in the ATSS and STSS model. We further show a stabilization time of $\Theta(n)$, a stabilization work of $\Theta(n)$, and a maintenance work of $\mathcal{O}(1)$.

3.1.1 Related Work

Maintaining overlay networks has already been considered in a large body of literature. In the past the work focused on how to keep an overlay network in a legal state, i.e. the network is already in a legal state before some changes happen and has to recover to a legal state afterward. Far less work focused on

self-stabilizing overlays, i.e. to recover to a legal state from any initial state.

In the case of structured Peer-to-Peer networks as CAN, Viceroy, Pastry, Skip graphs, SkipNet or Chord [69, 50, 70, 2, 26, 75] mechanisms to keep the overlay network in a legal state are already given in the introductory papers.

In [43] some stronger results could be shown for a specific overlay network as nodes were allowed to fail even in not fully repaired states of the network. In particular an adversary can at most add and/or remove $\mathcal{O}(\log n)$ peers in a constant time interval while the system remains functional. Nevertheless also this approach is not truly self-stabilizing as the set of states the system can recover from is limited, e.g. all remaining nodes have the same estimation of the total number of nodes in the system and are organized in supernodes forming a hypercube.

The idea of self-stabilization in distributed computing first appeared in a classical paper by E.W. Dijkstra in 1974 [17] in which he looked at the problem of self-stabilization in a token ring. Interestingly, though self-stabilizing distributed computing has received a lot of attention for many years, the problem of designing self-stabilizing networks has attracted much less attention.

In order to recover certain network topologies from any weakly connected network, researchers have started with simple line and ring networks. In [73], for example, Shaker and Reeves for the first time present a self-stabilizing protocol for an overlay network. To build a ring given by the order of the identifiers of the nodes each node searches for those nodes that succeed its own identifier by initializing search messages in the system that are always routed to the node with the closest succeeding identifier. We will use similar ideas in our protocols. In particular we use such message to build topologies that are not locally checkable. In [60], Onus et al. present a local-control strategy called linearization for converting an arbitrary connected graph into a sorted list. As we will see this protocol is a starting point for the protocol we will introduce to form a sorted list, which again is the basis for advanced self-stabilizing overlay networks.

In [19] Dolev and Kat describe a self-stabilizing protocol to build a hypertree with a polylogarithmic degree and search time. Their hypertree overlay network provides a bounded in- and out-degree of $\mathcal{O}(\log_b N)$ and a depth of also $\mathcal{O}(\log_b N)$, where N is the maximal number of nodes and b is an integer parameter > 1. The self-stabilizing protocol is based on three ideas. Firstly each node has to know the root. Additionally each node permanently ensures the validity of the its pointers to its parent and children. If the parent pointer is incorrect it then contacts the root. All other nodes are informed by the nodes by a broadcast and then collect information about the structure of their sub-trees in a convergecast such that in the end all information is collected by the root. The root then ensures that all nodes are assigned to their correct parental nodes and that the number of levels is not too high and otherwise enforces all nodes to contact the root again.

Jacob et al. [30] generalize insights gained from graph linearization to two dimensions and present a self-stabilizing construction for Delaunay graphs. In another paper, Jacob et al. [29] present a self-stabilizing variant of the skip graph and show that it can recover its network topology from any weakly connected state in $\mathcal{O}(\log^2 n)$ communication rounds with high probability. In [20] and [21] Dolev and Tzachar show self-stabilizing algorithms for forming sub-graphs like clusters or expanders in just polylogarithmic number of rounds.

In [7] the authors present a general framework for the self-stabilizing construction of overlay networks, which may involve the construction of the clique. The algorithm requires the knowledge of the 2-hop neighborhood for each node. In that way, failures at the structure of the overlay network can easily be detected and repaired. However, the work in order to do that when using this method is high as in each round a node sends the information about its complete neighborhood to all its neighbors.

3.2 The Sorted List

3.2.1 Introduction

The first overlay problem for which we present a solution is the sorted list problem. We denote this overlay problem by $OP = LIST$. A sorted list is given if each node is connected to its preceding and succeeding node in an order established on the nodes $v_0, v_1, \cdots, v_{n-1}$. See Figure 3.1 for an example. A list topology is probably the most simple topology one can think of. Therefore we start considering this simple topology in the hope that also the solving protocol is simple and easy to understand to give a first example for topological self-stabilization.

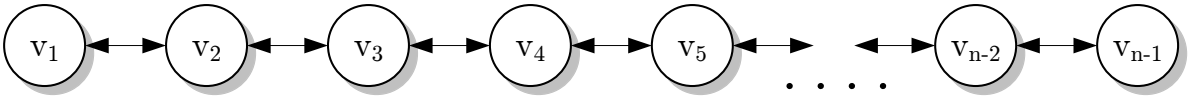


Figure 3.1: An example of a sorted list

3.2.2 Our Contribution

We present a protocol P^{LIST} that solves the overlay problem $LIST$. We show the correctness of P^{LIST} in the ATSS model in Theorem 3.2.1. The basic ideas of the protocol P^{LIST} are given in the paper [60]:

Melih Onus and Andrea W. Richa and Christian Scheideler, Linearization: Locally Self-Stabilizing Sorting in Graphs, ALENEX, 2007

Our results of a stabilization time of $\Theta(n)$ of P^{LIST} in the STSS model given in Theorem 3.2.5 is based on proofs we firstly showed in [35], which is a joint work with my colleague Andreas Koutsopoulos and our supervisor Christian Scheideler:

Sebastian Kniesburges and Andreas Koutsopoulos and Christian Scheideler, Re-Chord: a self-stabilizing chord overlay network, SPAA, 2011

The results for the stabilization work given in Theorem 3.2.7, the maintenance work given in Theorem 3.2.8, the work due to a single join given in Theorem 3.2.9 and in particular the improved stabilization work of a modified protocol $P^{LISTsync}$ given in Theorem 3.2.10 are presented for the first time.

3.2.3 Formal Definition

According to our model presented in Chapter 2 we assume that each node can be identified by an unique id. For a node v we define a hash function $h : ID \mapsto [0, 1]$ that maps the identifier of a node to a point $h(v)$ in the $[0, 1]$ interval. We refer to $h(v)$ as v's position. We assume that h is chosen in a way such that no two nodes get the same position. We further assume that h is known to every node, i.e. by knowing the identifier of one node each node can determine its position. Then a global order on the nodes can be defined by their positions $h(v_0) < h(v_1) < \cdots < h(v_{n-1})$ and each node can establish an order in the

same way on its local neighborhood, which enables nodes to check whether the program state is globally correct.

We can now define the sorted list as a list of nodes built according to the order of their positions.

Definition 3.2.1 *A graph $G = (V, E)$ is* a sorted list*, if $V = \{v_0, \cdots v_{n-1}\}$ with $h(v_i) < h(v_i + 1) \forall i \in \{0, \cdots n-1\}$ and $E = \{(v_i, v_j) :, j = i - 1 \vee j = i + 1 \forall 0 < j < n - 1\}$.*

We now define the internal variables of each node used in our protocol P^{LIST} to solve the overlay problem $LIST$. Each node u stores the following variables:

- *u.predecessor*: the preceding neighbor of u, i.e. a node v such that $h(u.predecessor) < h(v)$.
- *u.successor*: the succeeding neighbor of u, i.e. a node v such that $h(u.successor) > h(v)$.

Additionally each node contains the following variables:

- *u.Ch*: the implementation of $Ch(u)$, a buffer storing all incoming messages.
- *u.τ*: a boolean variable that is periodically true to activate the periodic actions at the node u.

To communicate with each other, nodes send messages. In our protocol there is only one type of message. The only content of a message is an identifier of some node. Thus a message m is of the following form $m = (v)$. In the following we define when an assignment of the variables is valid and how the sets IT and $LIST$ look like.

Definition 3.2.2 *An assignment of the variables of a node u is* valid *if $h(u.predecessor) < h(u)$ or $u.predecessor = nil$ and $h(u.successor) > h(u)$ or $u.successor = nil$.*

Note that an invalid assignment can be locally repaired immediately. Thus we assume in the following w.o.l.g. that initially the assignment is valid for every node.

With the given internal variables we are now able to define the set of initial topologies IT and especially the set of goal topologies $LIST$.

Definition 3.2.3 *Let E_e and E_i be defined as described in Chapter 2 according to the definition of internal variables. Then the* set of initial topologies *is given by:*

$$IT = \{G = (V, E = E_e \cup E_i) : G \text{ is weakly connected}\}$$

Definition 3.2.4 *Let E_e and E_i be defined as described in Chapter 2. Then the* set of target topologies *is given by:*

$$LIST = \{G = (V, E) : G_e \text{ is a sorted list}\}$$

3.2.4 Protocol P^{LIST}

In this section we give a description of the used protocol for the topological self-stabilization of a sorted list. The protocol is executed in a distributed way, i.e. each node executes the same protocol based on its local information. We assume that the initial state and every following program state is valid, i.e. the variables of each node are valid. The protocol P^{LIST} is based on the following simple idea firstly presented in [60]. Each node performs a local sorting on the positions of nodes received by messages

and of nodes stored in internal variables. We firstly describe the protocol informally and then give an implementation in pseudo code.

We assume a node u receives a message $m = (v)$ with the identifier of a node v. Then u checks whether $h(v) > h(u)$ or $h(v) < h(u)$. If $h(v) > h(u)$ and $h(v) < h(u.successor)$ or $u.successor = nil$, then u sends a message $m = (w = u.successor)$ to v (only if $u.successor \neq nil$) and updates its internal variable $u.successor$ and sets it to v, i.e. $u.successor = v$ in the next state. If $h(v) > h(u)$ and $h(v) > h(u.successor)$ then u sends a message $m = (v)$ to $w = u.successor$. In such a case we say that u *forwards* v to w. An illustration of the described actions is given in Figure 3.2.

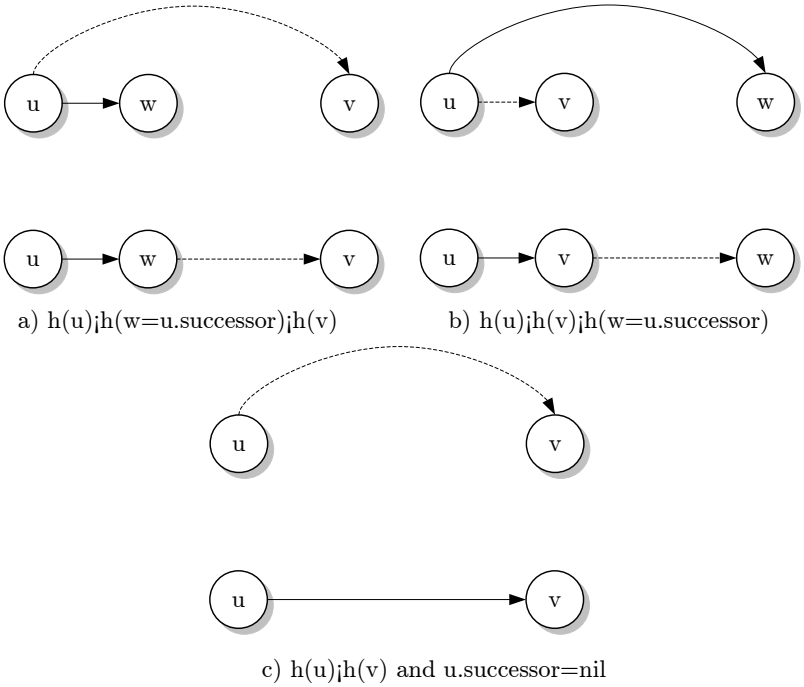


Figure 3.2: Three possible cases if a node u receives a node v

If $h(v) < h(u)$ and $h(v) > h(u.predecessor)$ or $u.predecessor = nil$, then u sends a message $m = (w = u.predecessor)$ to v (only if $u.predecessor \neq nil$) and updates its internal variable $u.predecessor$ and sets it to v, i.e. $u.predecessor = v$ in the next state. If $h(v) < h(u)$ and $h(v) < h(u.predecessor)$ then u sends a message $m = (v)$ to $w = u.predecessor$.

In this way nodes are sorted according to their positions and a node forwards all nodes it does not store in internal variables to nodes with a closer position.

Additionally each node u periodically informs its closest neighbors (stored in $u.successor$ and $u.predecessor$) about itself by sending a message $m = (u)$. This is done to achieve a strongly connected list. An illustration of the periodic action is given in Figure 3.3.

So there are two different kinds of actions, one is executed if a node receives a message and the other is executed periodically. To describe this formally we need two guards. The first guard is given by message $m \in u.Ch \rightarrow \cdots$. For the second guard we use the defined boolean predicate $u.\tau \rightarrow \cdots$ that is periodically true. We now give an pseudo code implementation of the protocol P^{LIST} in Algorithm 3.2.1.

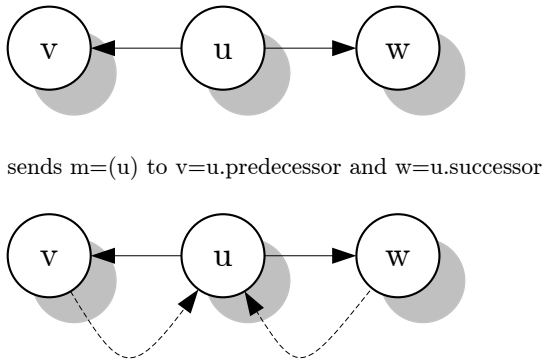


Figure 3.3: Outcome of the periodic action of a node u

Algorithm 3.2.1 P^{LIST}

message $m = (v) \in u.Ch \rightarrow$
if $h(v) > h(u)$ **then**
  if $u.successor = nil$ **then**
    $u.successor = v$
  else if $h(v) < h(u.successor)$ **then**
    send message $m' = (u.successor)$ to v
    $u.successor = v$
  else if $h(v) > h(u.successor)$ **then**
    send message $m' = (v)$ to $u.successor$
else
  if $u.predecessor = nil$ **then**
    $u.predecessor = v$
  else if $h(v) > h(u.predecessor)$ **then**
    send message $m' = (u.predecessor)$ to v
    $u.predecessor = v$
  else if $h(v) < h(u.predecessor)$ **then**
    send message $m' = (v)$ to $u.predecessor$
$u.\tau \rightarrow$
send message $m = (u)$ to $u.successor$
send message $m' = (u)$ to $u.predecessor$

3.2.5 Analysis in the ATSS model

In this section we analyze the protocol P^{LIST} in the ATSS-model. We show the correctness in an asynchronous setting by proving the convergence and closure property separately and then combine the results to prove the following main theorem:

Theorem 3.2.1 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{LIST} then eventually the graph converges to a graph $G' \in LIST$ (Convergence). If in an initial graph $G \in LIST$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in LIST$ (Closure).*

Convergence

We begin by showing that P^{LIST} fulfills the convergence property.

Theorem 3.2.2 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{LIST} then eventually the graph converges to a graph $G' \in LIST$.*

To show the theorem we first prove a set of lemmas from which the theorem follows. The main idea of the proofs is that the *lengths of the edges*, measured by the number of nodes between the start and endpoint in the sorted list, decrease until all consecutive nodes in the sorted list are directly connected.

Lemma 3.2.1 *If a computation of P^{LIST} started in a state in IT contains a state s_t where nodes u, v are connected by a path in G^t then in every state $s_{t'}$, $t' > t$, after that there is a path from u to v in $G^{t'}$.*

Proof. We show that in the executed actions according to the protocol P^{LIST} edges are kept, added or substituted by a path, such that the connectivity is maintained and no paths are disconnected. We therefore assume that $(u, v) \in E^t$ and $(u, v) \notin E^{t+1}$:

1. $(u, v) \in E_e^t$: If $v = u.successor$ then u receives a message $m = (w)$ with $h(u) < h(w) < h(v)$ and u updates its successor and forwards v to w. Then $(u, w) \in E_e^{t+1}$ and $(w, v) \in E_i^{t+1}$ and u and v stay connected. If $v = u.predecessor$ then u receives a message $m = (w)$ with $h(u) > h(w) > h(v)$ and u updates its predecessor and forwards v to w. Then $(u, w) \in E_e^{t+1}$ and $(w, v) \in E_i^{t+1}$ and u and v stay connected.

2. $(u, v) \in E_i^t$: Then u receives a message $m = (v)$. If $h(v) > h(u)$ and $h(v) > h(u.successor)$ the edge (u, v) in G^t is replaced by a path $u, w = u.successor, v$ in G^{t+1}. The edge (u, w) obviously exists as u currently stores $w = u.successor$ and the edge $(w, v) \in E_i^{t+1}$ is created by a message $m = (v)$ sent to w. If $u.successor = nil$ the edge (u, v) is retained in G^{t+1} and even G_e^{t+1}. If $h(u.successor) > h(v) > h(u)$ then u stores v as its successor and thus $(u, v) \in E_e^{t+1}$. The same arguments hold for the case $h(v) < h(u)$.

In the periodic action edges are only added to G^t, as u introduces itself to its predecessor and successor. □

The next lemma claims that the stored edges $(u, u.successor)$ and $(u, u.predecessor)$ are shortened over time.

Lemma 3.2.2 *If the computation of P^{LIST} reaches a state s_t where for some node u there are two edges $(u,v) \in E_e^t$ and $(u,w) \in E_i^t$ such that $h(u) < h(w) < h(v)$ (resp. $h(v) < h(w) < h(u)$) then this computation contains a later state $s_{t'}$, $t' > t$, with an edge $(u,w') \in E_e^{t'}$ with $h(w') \leq h(w)$ (resp. $h(w') \leq h(w)$).*

Proof. We only show the case $h(u) < h(w) < h(v)$. For $h(u) > h(w) > h(v)$ the same arguments can be applied. If $(u,w) \in E_i^t$ then there is a message $m = (w) \in u.Ch$. Once this message is received at time t' by u, $u.successor$ is either updated to w, such that $(u,w') \in E_e^{t'}$ with $w' = w$, or if $u.successor$ is not updated, $u.successor$ must already be set to w', such that $(u,w') \in E_e^{t'}$ with $h(w') < h(w)$. □

The next lemma claims that edges given by a channel state are also shortened over time, but with the difference that for these links the start-point of the link and not its endpoint is updated.

Lemma 3.2.3 *If the computation of P^{LIST} reaches a state s_t where for a node u there are links $(u,v) \in E_e^t$ and $(u,w) \in E_i^t$ with $h(u) < h(v) < h(w)$ (resp. $h(u) > h(v) > h(w)$) then the computation contains a later state $s_{t'}$, $t' > t$, where there is a link $(w',w) \in E_i^{t'}$ with $h(u) < h(w') < h(w)$ (resp. $h(u) > h(w') > h(w)$).*

Proof. We again only show the case $h(u) < h(v) < h(w)$. For $h(w) < h(v) < h(u)$ the same arguments can be applied. If $(u,w) \in E_i^t$ there is a message $m = (w) \in u.Ch$. Once this message is received by u, we know by Lemma 3.2.1 that there is a state $s_{t'}$ at time $t' > t$ where there is an edge $(u,x) \in E_e^{t'}$ with $h(u) < h(x) \leq h(v)$ as the stored edges in E_e are only shortened over time. Then w is forwarded in state s_{t^*}, $t^* > t'$, to $w' = u.successor$ with $h(w') \leq h(x)$ by a message $m' = (w)$. It then holds that $h(u) < h(w') < h(w)$ and $(w',w) \in E^{t^*}$. □

Lemma 3.2.4 *If the computation of P^{LIST} reaches a state s_t where for some nodes u, v and w such that $h(u) < h(w) < h(v)$ (resp. $h(v) < h(w) < h(u)$) there are edges $(u,v) \in E_e^t$ and $(w,u) \in E_e^t$ then the computation contains a later state $s_{t'}$, $t' > t$, where either some edge in $E_e^{t'}$ is shorter than in E_e^t, i.e. the startpoint is closer to the endpoint of the edge at time t' than at time t, or $(u,w) \in E_e^{t'}$.*

Proof. Remember that the periodic action is periodically enabled. In this action w sends a message $m = (w)$ to u. Then by Lemma 3.2.2 this lemma is proven. □

Lemma 3.2.5 *If the computation of P^{LIST} reaches a state s_t where there is an edge $(u,v) \in E_e^t$ then the computation contains a later state $s_{t'}$, $t' > t$, where some edge in $E_e^{t'}$ is shorter than in E_e^t or $(v,u) \in E_e^{t'}$.*

Proof. W.l.o.g. let $h(u) < h(v)$. Eventually in a state $s_{t'}$ at time $t' > t$ u executes a periodic action and the edge (v,u) is added to $E_i^{t'}$ (if not already existing). If this edge is also added to $E_e^{t'}$, that is $v.predecessor = u$, then the lemma holds. Otherwise we assume that no edge in E_e^t shortens. Then $h(u) < h(w = v.predecessor) < h(v)$ and according to lemma 3.2.3 the edge $(v,u) \in E^{t'}$ is shortened to an edge $(w',u) \in E^{t^*}$ in a state s_{t^*} at time $t* > t'$ where $h(u) < h(w') < h(w)$. If $(w',x) \in E_e^{t^*}$ where $h(x) < h(u)$, then an edge in E_e^t has to shorten eventually according to Lemma 3.2.2. If $x = u$, Lemma 3.2.4 holds and again an edge in E_e^t has to shorten. Otherwise according to Lemma 3.2.3 the edge $(w',u) \in E^{t^*}$ to u shortens to (w'',u) where $h(u) < h(w'') < h(w')$. This implicit edge can only shorten a finite number of times until an edge in E_e^t has to be shortened. □

Lemma 3.2.6 *If the computation of P^{LIST} reaches a state s_t such that $(u,v) \in E_e^{t'} \Rightarrow (v,u) \in E_e^{t'}$ in every state $s_{t'}$, $t' > t$, then this computation contains a state s_{t*} such that E_e^{t*} is strongly connected.*

Proof. If $(u,v) \in E_e^{t'} \Rightarrow (v,u) \in E_e^{t'}$ holds for every $t' > t$ then in a periodic action no edges are added to E^t. Edges in E_i^t are shortened over time according to Lemma 3.2.3, such that eventually a state s_{t^*} is reached with $E_e^{t^*} = E^{t'}$. As E^t is weakly connected at time $t = 0$ and stays connected by Lemma 3.2.1 and $(u,v) \in E_e^t \Rightarrow (v,u) \in E_e^t$ holds, eventually a state s_{t^*} is reached such that $E_e^{t^*}$ is strongly connected. □

Lemma 3.2.7 *If the computation of P^{LIST} reaches a state s_t such that E_e^t is strongly connected and for every pair of nodes $(u,v) \in E_e^t \Rightarrow (v,u) \in E_e^t$ then this state is a solution for the sorted list problem and $G^t \in LIST$.*

Proof. Let us assume that it is not a solution for the sorted-list problem. Then there is a pair v_i, v_{i+1} of consecutive nodes that are not directly connected. As E_e^t is strongly connected there is a shortest path from v_i to v_{i+1} and vice versa. On this path there has to be a node u that has two outgoing edges to nodes v, w $h(v) < h(u)$ and $h(w) < h(u)$ (or $h(v) > h(u)$ and $h(w) > h(u)$). This is a contradiction to the definition of E_e^t in which each node has only one outgoing edge to its predecessor and successor. □

We are now ready to show the Theorem. According to Lemma 3.2.1 G^t stays weakly connected during the computation. By Lemma 3.2.2 and 3.2.3 it follows that all edges are shortened over time. Note that as soon as a node receives a message there has to be a state s_{t_1} at time t_1 with at least one edge in $E_e^{t_1}$. According to Lemma 3.2.4 eventually all edges in $E_e^{t_1}$ can not be shortened and either (u,v) and $(v,u) \in E_e^{t_2}$ in some state $t_2 \geq t_1$ or a new edge is added to $E_e^{t_1}$. As for each node there are at most two edges in E_e^t, eventually all edges are added and a state s_{t_3} at time $t_3 \geq t_2$ is reached with $(u,v) \in E^{t_3} \Rightarrow (v,u) \in E_e^{t_3}$. According to Lemma 3.2.5 there is a state such that in every state s_{t_4} at time $t_4 \geq t_3$ $(u,v) \in E_e^{t_4} \Rightarrow (v,u) \in E_e^{t_4}$. Now, according to Lemma 3.2.6 the computation contains a later state s_{t_5} at time $t_5 \geq t_4$ such that $E_e^{t_5}$ is strongly connected. Then by applying lemma 3.2.7 in this state s_{t_5} the sorted-list problem is solved.

Closure

We will now show that for P^{LIST} also the closure property holds, i.e. that once a legal state is reached in the computation only legal states are reachable. This concludes our main theorem.

Theorem 3.2.3 *If in an initial graph $G \in LIST$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in LIST$.*

Proof. If the computation reaches a state s_t such that G^t is a solution for the sorted list problem, then it follows from the definition that the graph is strongly connected and for every pair of consecutive nodes in the sorted list v_i, v_{i+1} with $h(v_i) < h(v_{i+1})$ $(v_i, v_{i+1}) \in E_e^t$ and $(v_{i+1}, v_i) \in E_e^t$. Then $E_e^{t+1} = E_e^t$ as no edges in E_e^t are deleted or added or forwarded. E_e^t only changes if a node receives a message containing a node that replaces either its successor or its predecessor. Obviously this can not happen if v_i and v_{i+1} are already connected. □

Combining Theorem 3.2.2 and Theorem 3.2.3 we get our main theorem Theorem 3.2.1 that states that P^{LIST} is a topological self-stabilizing protocol for the sorted list problem.

3.2.6 Analysis in the STSS model

Analyzing the same protocol P^{LIST} in the STSS model we show some bounds on the complexities defined in Chapter 2.

Theorem 3.2.4 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{LIST} then the graph converges to a graph $G' \in LIST$ (Convergence) with a stabilization time of $\mathcal{O}(n)$ and a stabilization work of $\Theta(n^2)$. If in an initial graph $G \in LIST^*$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in LIST$ (Closure) with a maintenance work of $\mathcal{O}(1)$.*

Stabilization Time

We start by showing the stabilization time of the protocol P^{LIST}. Note that in the $STSS$ model we execute all enabled actions in one round, i.e. all messages in $u.Ch$ are received by a node u. Thus we slightly adapt the notation of E^t and E^t_e. Instead of considering the changed edge sets after each single action we consider the changed edge sets after executing all enabled actions for all nodes in one round.

Theorem 3.2.5 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{LIST} then the graph converges to a graph $G' \in LIST$ (Convergence) with a stabilization time of $\Theta(n)$.*

To show the correctness of P^{List} and the stabilization time in the STSS model we introduce some further definitions and show a helpful lemma.

Definition 3.2.5 *A path $p = (u = w_0, w_1, \cdots w_l = v)$ is called a* weakly connecting path *for a pair of nodes u, v if $(w_i, w_{i+1}) \in E$ or $(w_{i+1}, w_i) \in E$ $\forall 0 \leq i \leq l$. Let $w_{min} = \min\{h(w_i) \in p\}$ and $w_{max} = \max\{h(w_i) \in p\}$. We call w_{min} and $w_{max}$the* border nodes *of p and $[h(w_{min}), h(w_{max})]$the* range *of the weakly connecting path p.*

Lemma 3.2.8 *If a pair of nodes u and v with $h(u) < h(v)$ (resp. $h(v) < h(u)$) is connected by a directed path $p = (u = w_0, w_1, \cdots w_l = v)$ from u to v over nodes w_i with $h(u) \leq h(w_i) \leq h(v)$ (resp. $h(v) \leq h(w_i) \leq h(u)$) at a time t then there is directed path $p' = (u = w'_0, w'_1 = u.successor, w'_2, \cdots w'_l = v)$ from u to v over nodes w'_i with $h(u) \leq h(w'_i) \leq h(v)$ (resp. $h(v) \leq h(w'_i) \leq h(u)$) at every time $t' > t$.*

Proof. We show that each directed edge $(u, v) \in E^t$ either stays or is substituted by such a directed path from u to v. W.l.o.g. we assume $h(u) < h(v)$. If there is an edge $(u, v) \in E^t$ there can be two cases, such that $(u, v) \notin E^{t+1}$. Either $v = u.successor$ and u receives a message $m = (w)$ with $h(u) < h(w) < h(v)$ such that w becomes u's successor and a message $m' = (v)$ is sent to w or $(u, v) \in E^t_i$ and u receives the message containing v and $h(u.successor) < h(v)$ and a message $m' = (v)$ is sent to $w = u.successor$. In both cases (u, v) is substituted by a directed path $(u, w), (w, v)$ as $(u, w) \in E^{t+1}_e$ and $(w, v) \in E^{t+1}_i$ and $h(u) < h(w) < h(v)$. In each other case the edge (u, v) remains with $v = u.successor$. □

We are now ready to prove Theorem 3.2.5.

Proof. We first show that the stabilization time is in $\mathcal{O}(n)$. Let v_i and v_{i+1} be two arbitrary consecutive nodes in the sorted list. By the definition of IT we know that in every graph $G \in IT$ there is a weakly

connecting path for v_i and v_{i+1}. We show that the minimal range of all connecting paths is decreasing, such that the two nodes are directly connected after at most $\mathcal{O}(n)$ synchronous rounds.

Let p^t be a connecting path for two arbitrary consecutive nodes v_i and v_{i+1} at time t. We then show that we can find a connecting path at time t', $t' > t$, with a strictly smaller range. We firstly prove that if one of the border nodes w^t_{min} and w^t_{max} defining the range of p_t has two outgoing edges in p^t, then we can construct a connecting path p^{t+1} with a smaller range. W.l.o.g. we show this for w^t_{min} as the same arguments can be applied for w^t_{max}.

If w^t_{min} has two outgoing edges $(w^t_{min}, x), (w^t_{min}, y) \in p^t$ then in the next round according to Lemma 3.2.8 there is a directed path $p_1 = (w^t_{min} = w'_0, w^t_{min}.successor = w'_1, w'_2, \cdots w'_l = x)$ from w^t_{min} to x such that $h(w^t_{min}) \leq h(w'_i) \leq h(x) \forall w'_i$ and another path $p_2 = (w^t_{min} = w''_0, w_{min}.successor = w''_1, w''_2, \cdots w''_k = y)$ from w^t_{min} to y such that $h(w^t_{min}) \leq h(w''_i) \leq h(y) \forall w''_i$. Thus we can substitute the edges (w^t_{min}, x) and (w^t_{min}, y) by p_1 and p_2 leaving out w^t_{min}, as p_1 and p_2 both include $w^t_{min}.successor$ and get a new weakly connecting path p_{t+1} with a strictly smaller range (see Figure 3.4).

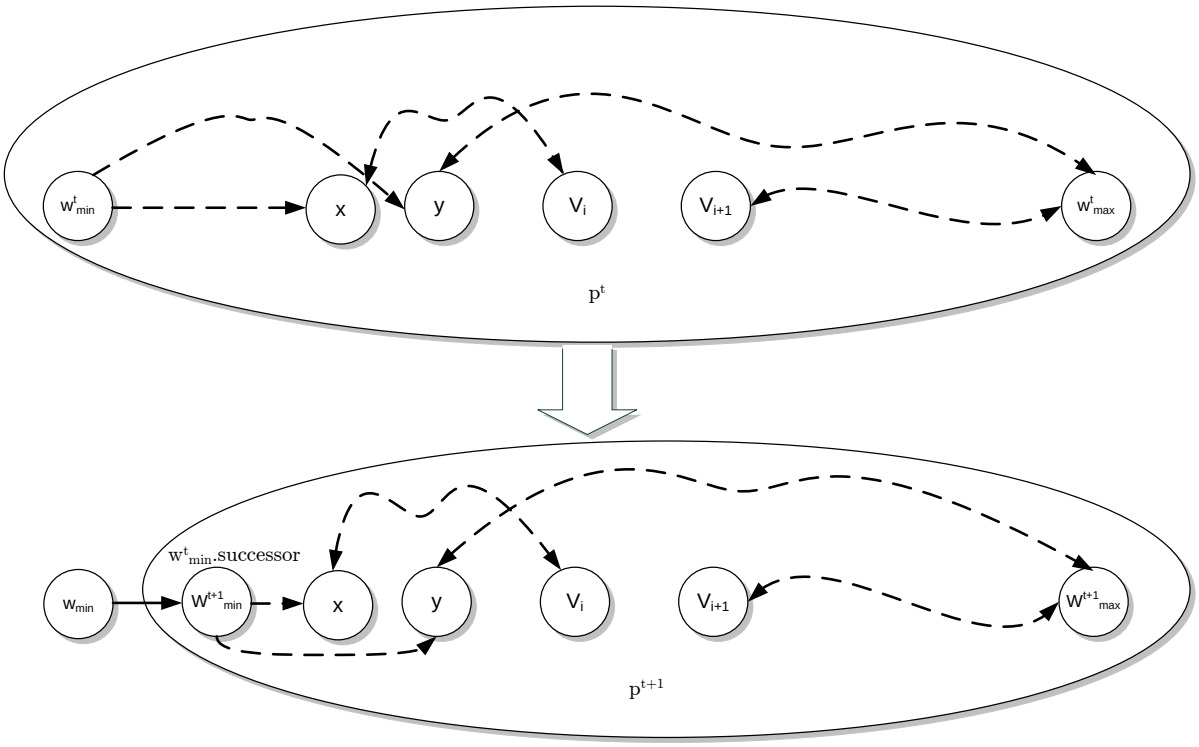


Figure 3.4: The range of p^t decreases if w^t_{min} has two outgoing edges

It remains to show that it does not take too long before a border node has two outgoing edges. In case v_{min} does not have two outgoing edges we construct p^{t+1} out of p^t in the following way. If $(x, y) \in p^t$ with $h(x) < h(y)$ and y is delegated then according to Lemma 3.2.8 there is a directed path $p' = (x = w_0, w_1 = x.successor, w_2, \cdots w_l = y)$ with $h(x) \leq h(w_i) \leq h(y)$ from x to y. Thus we substitute (x, y) by p' in p^t to get p^{t+1}. If $(x, y) \in p^t$ with $h(x) < h(y)$ and y is not delegated then we simply keep $(x, y) \in p^{t+1}$.

If $(x, y) \in p^t$ with $h(x) > h(y)$ and y is delegated then according to Lemma 3.2.8 there is a directed path $p' = (x = w_0, w_1 = x.predecessor, w_2, \cdots w_l = y)$ with $h(x) \geq h(w_i) \geq h(y)$ from x to y. Thus

we substitute (x, y) by p' in p^t to get p^{t+1}. If $(x, y) \in p^t$ with $h(x) > h(y)$ and y is not delegated, then x introduces itself to y and we substitute $(x, y) \in p^t$ by $(y, x) \in p^{t+1}$. Note that by all substitutions the range of the path is not increasing. By using this construction scheme we can show that for a path p^t it takes at most $\max\{0, n-t\}$ rounds before w^t_{min} has two outgoing edges. We denote by w^t_{min} the left border node of the connecting path p^t.

We prove this by induction on t. Let p^0 be an arbitrary undirected path connecting two consecutive nodes v_i, v_{i+1} in the sorted list. Then w^0_{min} is the left border node of this path. If w^0_{min} already has two outgoing edges on p^0 then it takes obviously at most $\max\{0, n\}$ rounds. Otherwise there is at least one incoming edge (y, w^0_{min}) on p^0. For each such incoming edge (y, w^0_{min}) the node y either introduces itself to w^0_{min} or delegates w^0_{min}. In the first case w^0_{min} then has an outgoing edge in p^1 according to our construction scheme. In the second case (y, w^0_{min}) is substituted by a path p' as described above. Then w^0_{min} still has an incoming edge (y', w^0_{min}), but with $h(y') < h(y)$. Obviously w^0_{min} can only be delegated n times before (y', w^0_{min}) is substituted by an outgoing edge. Let p^t be an undirected path connecting v_i, v_{i+1} in the sorted list that was constructed out of p^0 according to our construction scheme. If w^t_{min} already has two outgoing edges on p^t then it takes obviously at most $\max\{0, n-t\}$ rounds. Otherwise there is at least one incoming edge (y, w^t_{min}) on p^t. Then $(y, w^t_{min}) \notin p^{t-1}$ but w^t_{min} was delegated to y by another node z with $h(z) > h(y)$, otherwise y would have introduced itself to w^t_{min} and w^t_{min} would have an outgoing edge instead. Thus $(z, w^t_{min}) \in p^{t-1}$. For this edge the same observation holds. Then w^t_{min} has been delegated t times by nodes z with $h(z) > h(y)$. Thus there are at most $n - t$ nodes w with $h(w^t_{min}) < h(w) < h(y)$ w^t_{min} can still be delegated to before the edge is substituted by an outgoing edge.

Then it takes at most $\max\{0, n-t\}$ rounds before w^t_{min} has two outgoing edges. If w^t_{min} has two outgoing edges we can construct a path with a smaller range. Then after $\mathcal{O}(n)$ rounds v_i, v_{i+1} are directly connected,i.e. $(v_i, v_{i+1}) \in E_e$ and $(v_{i+1}, v_i) \in E_e$. This gives us an upper bound of $\mathcal{O}(n)$ rounds for the stabilization time.

Consider the graph shown in Figure 3.5.

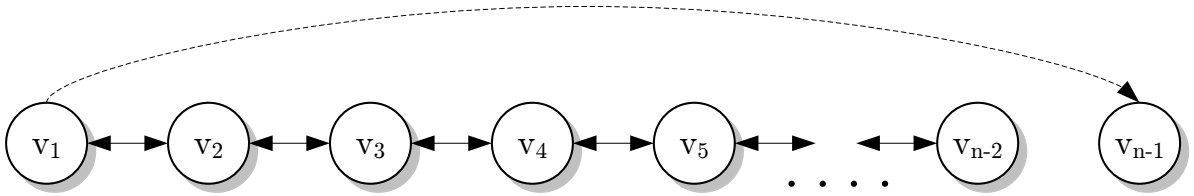


Figure 3.5: A sorted list with a stabilization time $\Theta(n)$

Then it obviously takes $\Omega(n)$ rounds until v_{n-1} is forwarded to v_{n-2} and v_{n-1} and v_{n-2} are introduced to each other, as in each round v_{n-1} is forwarded to one node closer to v_{n-2}. Thus in total we showed a stabilization time of $\Theta(n)$.

□

Theorem 3.2.6 *If in an initial graph $G \in LIST$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in LIST$ (Closure).*

Proof. The proof is the same as in the asynchronous setting. □

Stabilization Work

We proceed by giving a bound on the number of messages each node has to send and receive during the stabilization time. For the protocol P^{LIST} only a trivial bound of $\Theta(n^2)$ stabilization work can be shown.

Theorem 3.2.7 *If in an initial graph $G \in IT$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in LIST$ with a stabilization work of $\Theta(n^2)$.*

Proof. We first show that the stabilization work is in $\mathcal{O}(n^2)$. As we know the stabilization time is $\mathcal{O}(n)$ and in each of these rounds a node can receive at most $\mathcal{O}(n)$ different messages in each round. To avoid counting messages resulting from overfull channels in the initial state we assume that identical messages are only processed once in one round, i.e. identical messages in a channel are merged to one.

Additionally each node also sends only $\mathcal{O}(n)$ messages, as in each executed action at most $\mathcal{O}(1)$ messages are sent. Thus each node sends and receives at most $\mathcal{O}(n^2)$ messages until a legal state is reached. It remains to show that in the worst case $\Omega(n^2)$ messages are in fact sent or received by at least one node during the stabilization.

Therefore consider the following example given in Figure 3.6 of an initially weakly connected network.

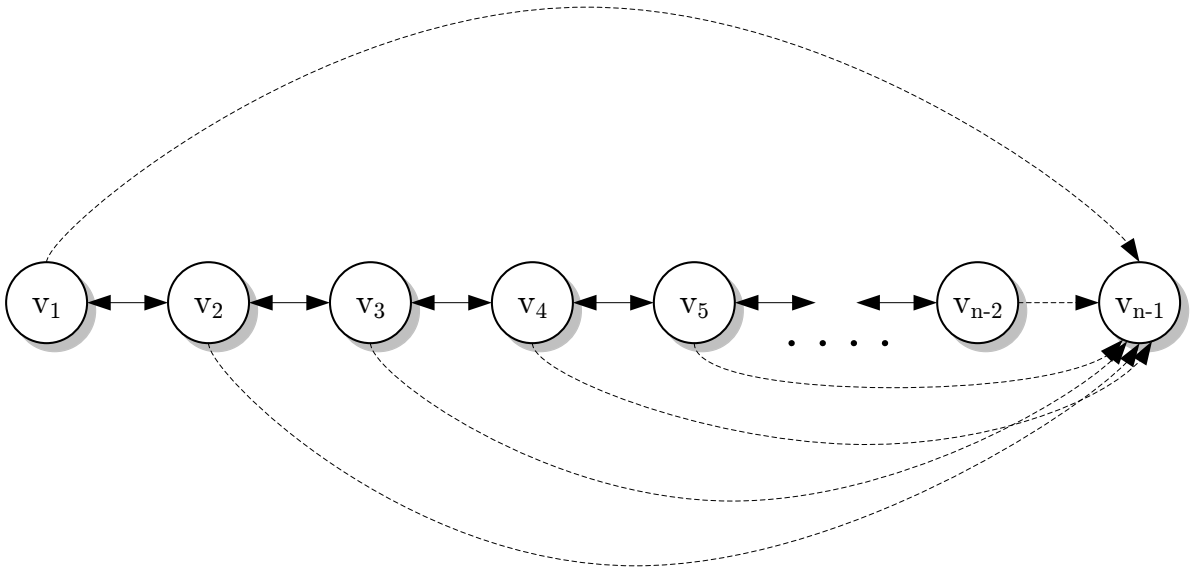


Figure 3.6: A sorted list with a stabilization work $\Theta(n^2)$

Then v_{n-1} is the successor of any other node in the network. Thus each node introduces itself to v_{n-1}, i.e. v_{n-1} receives $n-1$ messages. v_{n-1} forwards all received ids to its predecessor v_{n-2}. In the next round v_{n-2} forwards all received ids $(0, 1, 2, \cdots v_{n-4})$ to its predecessor v_{n-3} and introduces itself. Then v_{n-2} becomes the successor of v_{n-3} and v_{n-3} stops sending its identifier to v_{n-1}. So v_{n-1} receives $n-2$ messages in the next round. v_{n-3} then forwards all received ids $(0, 1, 2, \cdots v_{n-5})$ to its predecessor v_{n-4} and will become its successor. So then there are only $n-3$ nodes for which v_{n-1} is the current successor. Applying this argument inductively follows that after $n-1$ rounds only v_{n-2} stores v_{n-1} as its successor and a legal state is reached. During this stabilization v_{n-1} received $n-1+\sum_{i=1}^{n-1} i = \Omega(n^2)$ messages. □

Maintenance Work

Theorem 3.2.8 *If in an initial graph $G \in LIST^*$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in LIST^*$ with a maintenance work of $\mathcal{O}(1)$.*

Proof. In any graph $G \in LIST^*$ there are only edges $(v_i, v_{i+1}), (v_{i+1}, v_i) \in E_e^t$ and edges $(v_i, v_{i+1}), (v_{i+1}, v_i) \in E_i^t$ for a pair of consecutive nodes in the sorted list v_i, v_{i+1}. Thus a node only sends $O(1)$ messages in each round by introducing itself to its predecessor and successor and receives only $\mathcal{O}(1)$ messages in each round by its predecessor and successor. □

Single Join Events

For the completeness of the analysis we consider dynamic changes of the network in terms of single joining nodes. For the sorted list we don't analyze the case of a leaving node, as leaving is trivial for the nodes with the minimal and maximal position v_0 and v_{n-1} and all other nodes are not allowed to leave to preserve connectivity.

Theorem 3.2.9 *If a node joins a network $G \in LIST$ it takes at most $\mathcal{O}(n)$ rounds until a legal state is reached again with an additionally work of $\mathcal{O}(1)$ for each node per round.*

Proof. A node u joins an existing network at time t by being aware of the identifier of exactly one node v already in the network. u like the nodes already in the network execute the protocol P^{LIST}. Thus u will introduce itself to v as $v = u.successor$ if $h(u) < h(v)$ or $v = u.predecessor$ otherwise. For the rest of the proof we assume $h(u) > h(v)$, in the other case symmetric arguments can be applied. Then after one round $(u, v) \in E_i^{t+1}$. According to P^{LIST} v forwards u to $v.successor$ if $h(u) > h(v.successor) > h(v)$ or stores u as $v.successor$. As u can only be forwarded $\mathcal{O}(n)$ times, after $l \in \mathcal{O}(n)$ rounds $(v', u) \in E_i^{t+l}$ with $h(v') < h(u) < h(w = v'.successor)$. Thus u becomes $v'.successor$ and the old node's identifier w is forwarded to u. As the network was in legal state before u stores w correctly as $u.successor$. After one further round of introduction a new legal state is reached. During this $\mathcal{O}(n)$ rounds u introduced itself to v in each round, thus it takes additionally $\mathcal{O}(n)$ rounds until a state in $LIST$ is reached. In each round u only introduces itself to v and all other nodes already in the network only send and receive introduction messages to and from their successor and predecessor and one message containing u. So each node sends and receives at most $\mathcal{O}(1)$ messages in each round. □

3.2.7 Modification

As we have seen the protocol P^{LIST} works correctly for an asynchronous and synchronous setting. In the following we improve the performance in a synchronous setting, in particular the stabilization work, by exploiting the fact that all nodes execute the protocol $P^{LISTsync}$ synchronously and especially that all messages are received at once in the beginning of one round.

The way to reduce the stabilization work is to avoid sending unnecessary messages. One way to achieve this is to respond to the periodic messages sent by nodes that assume that they are the successor or predecessor of the receiving node u by not only forwarding the identifier to u's current successor or predecessor but also to introduce u's successor or predecessor to the sending node. By this modification u will not receive periodic messages of these nodes again. To distinguish these periodic messages from other messages we extend our messages by a message type. So messages now look like $m = (type, id)$.

We use two types one for for the periodic introduction messages *introduction* and one for the forwarding messages *forward*.

Another way is to exploit the fact that all messages are received at once, so instead of treating each *forward* message individually we can handle them as a batch. Then we can sort all ids received by messages and forward them to their neighbor in this sorting instead of forwarding each identifier to u' successor or predecessor. E.g. if u receives the ids $v_1, v_2, \cdots, v_k$ with $h(u.successor) < h(v_i) < h(h_{i+1})$, then u will forward v_{i+1} to v_i and only v_1 to $u.successor$. Thus each node only receives one message in the next round instead of $\mathcal{O}(n)$ messages that would be sent to $u.successor$ in the original protocol P^{LIST}. An illustration is given in Figure 3.7.

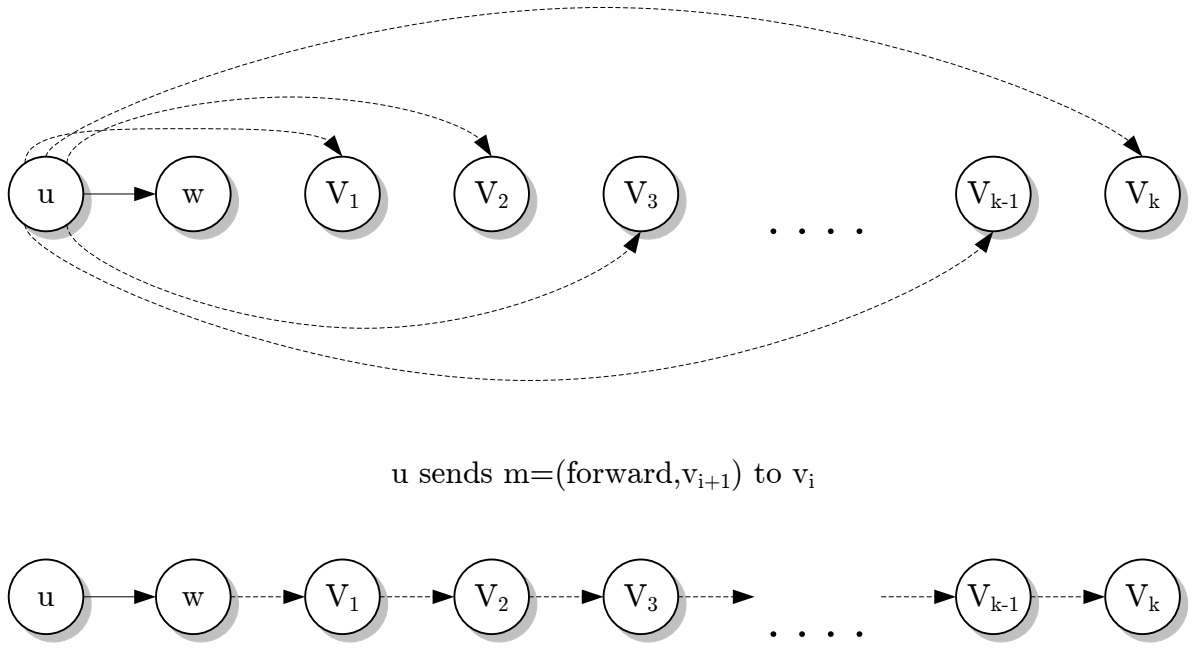


Figure 3.7: Forwarding in $P^{LISTsync}$

Then we get the following protocol $P^{LISTsync}$ presented in Pseudo-Code 3.2.2.

For this modified protocol we can show that the analysis for the correctness in the ATSS model still holds. In the ATSS model we only receive one message in each time step, and thus P^{LIST} and $P^{LISTsync}$ are the same executed in the ATSS model with the only difference that we not only forward a node to the predecessor or successor but might also forward the predecessor or successor to this node, so only some implicit edges is added. If $P^{LISTsync}$ is executed in the STSS model we can still apply our analysis of the stabilization time of $\Theta(n)$ rounds as we can still show that the range of a weakly connecting path shrinks to a direct edge in $\mathcal{O}(n)$ rounds.

However we are now able to show that $P^{LISTsync}$ has a stabilization work of $\Theta(n)$.

Theorem 3.2.10 *If in an initial graph $G \in IT$ every node executes the protocol $P^{LISTsync}$ then each possible computation leads to a graph $G^{LIST} \in LIST$ with a stabilization work of $\Theta(n)$.*

Proof. We start by showing an upper bound of $\mathcal{O}(n)$. We handle the two types of messages separately. We firstly show that each node u receives at most $\mathcal{O}(n)$ *introduction* messages. This follows from the

Algorithm 3.2.2 $P^{LISTsync}$

message $m \in u.Ch \rightarrow$
Sort all incoming ids v in messages and $u.successor$ and $u.predecessor$ according to their position $h(v)$, such that $h(v_{-l}) < h(v_{-(l-1)}) < \cdots < h(v_{-1}) < h(u) < h(v_1) < \cdots < h(v_{k-1}) < h(v_k)$
$u.successor = v_1$
for i=1 to k-1 **do**
  Send message $m' = (forward, v_{i+1})$ to v_i
  if thenv_i is received by a message $m = (introduction, v_i)$
    send message $m' = (forward, v_{i-1})$ to v_i
$u.predecessor = v_{-1}$
for i=1 to l-1 **do**
  Send message $m' = (forward, v_{-(i+1)})$ to v_{-i}
  if thenv_{-i} is received by a message $m = (introduction, v_{-i})$
    send message $m' = (list, v_{-(i-1)})$ to v_{-i}
$u.\tau \rightarrow$
send message $m = (introduction, u)$ to $u.successor$
send message $m' = (introduction, u)$ to $u.predecessor$

observation that each node v sends at most one *introduction* message to u while u knows a node w with position between u and v, i.e. v is not u's predecessor or successor. If v sends an *introduction* message u will respond by forwarding v to w and vice versa. So in the next round v knows w and will store it or another node with a position between v and w as its predecessor or successor, so u will never be v's predecessor or successor again. Also u only sends $\mathcal{O}(n)$ *introduction* messages during the stabilization as it sends at most two *introduction* messages in each round.

We can also show that each node u receives at most $\mathcal{O}(n)$ *forward* messages. W.l.o.g. we show that a node u receives at most $\mathcal{O}(n)$ forward messages with identifiers w, such that $h(u) < h(w)$. There are two cases in which u receives such a message. Either u sends an introduction message to a node z with $h(w) < h(z)$ and received w in response or w is sent to u by a node y with $h(y) < h(u) < h(w)$ that simply forwards w to a closer node according to its local sorting. The first case can happen at most once. As soon as u receives w it will never introduce it self to such a node z with $h(w) < h(z)$. So we only have to consider the second case. Let's assume u receives a_i forward messages containing w_i from nodes y with $h(y) < h(u)$ where the w_is are sorted in descending order $h(w_i) > h(w_{i+1})$ according to their position. The a_i messages containing w_i can have two sources. Either w_i was stored initially in a node y with $h(y) < h(u)$ or such a node y received w_i as a response to an introduction message to a node z with $h(z) > h(w_i)$, see Figure 3.8.

We therefore split a_i into two components. Let b_i be the number of times u receives w_i because initially there is an edge $(y, w_i) \in E^0$ and over time w_i is forwarded to u. Let further c_i be the number of times u receives w_i, because y received w_i after an introduction message. We first can observe that $\sum_{i=1}^{l} b_i = \mathcal{O}(n)$. If a node y initially has edges (y, w_i) and (y, w_j) with $j > i$ then this w_i will never be forwarded to u, as it will be forwarded to a node v with $h(u) < h(w_j) \leq h(v) < h(w_i)$. So there have to be $\sum_{i=1}^{l} b_i$ different nodes y with $h(y) < h(u)$. There can be at most $\mathcal{O}(n)$ such nodes.

To give a bound on $\sum_{i=1}^{l} c_i$ we have to determine how many different w_i that will be forwarded to u a node y can receive as a response to an introduction message. Let's assume a node y sends w_i and w_j to

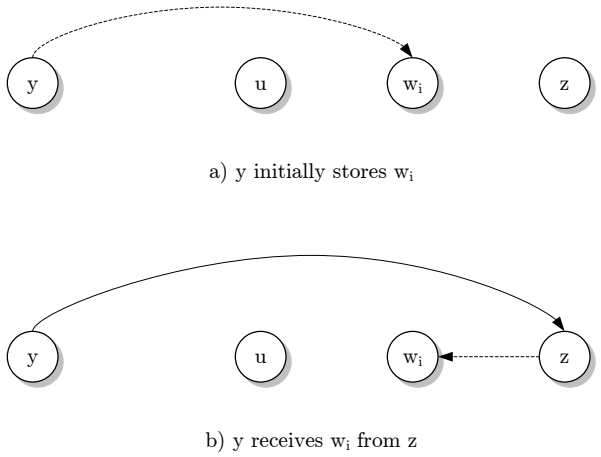


Figure 3.8: Different sources of w_i for a node y

u. If a node y receives w_i it can not have an edge (y, w_j) with $j > i$ as otherwise y would not send an introduction message to a node z with $h(z) > h(w_i) > h(w_j)$. So y receives w_j after w_i, but before y forwards w_i to a node v with $h(y) < h(v) < h(u)$, because otherwise y would not send an introduction message to a node z' with $h(z') > h(w_j) < h(v)$. So either y has edges to w_i and w_j at the same time and then forwards w_i to v' with $h(v') > h(u)$ and then w_i is never forwarded to u or w_i is forwarded to such a node v' before receiving w_j. In any case each node y receives at most one of the w_is as a response. Then $\sum_{i=1}^{l} c_i = \mathcal{O}(n)$. If a node receives at most $\mathcal{O}(n)$ forwarding messages it will also send at most $\mathcal{O}(n)$ forwarding messages. A forwarding message is sent because a node receives a forward message or a wrong introduction message, of both a node can receive at most $\mathcal{O}(n)$ many.

The lower bound follows directly from the lower bound of the stabilization time. As it takes at least $\Omega(n)$ rounds and at least one node sends at least $\Omega(1)$ messages in each round, as at least one node has a predecessor or successor in the initial state otherwise the initial graph would not be connected. □

3.3 Small-World Networks

3.3.1 Introduction

We now develop protocols for more complex networks like small-world networks. In the presented protocol we reuse the ideas of the protocol P^{LIST} and its analysis. We denote the overlay problem we want to solve in this section by $OP = SMALL - WORLD$. An example of a small-world network that we consider in this section is given in Figure 3.9.

So far we developed a model for topological self-stabilization and applied it to the sorted list as a simple example. Obviously the sorted list is not a network anyone would use in a setting where faults of links and processes are likely to happen. As soon as one node or link fails the whole networks becomes disconnected. Furthermore the sorted list does not provide properties like a small diameter or short routing lengths, which are in fact $\Theta(n)$. As we motivated the concept of topological self-stabilization by the observation that faults are likely in large unsupervised networks like peer-to-peer networks, we transfer the methodology introduced for the sorted list to more complex but also more robust and therefore

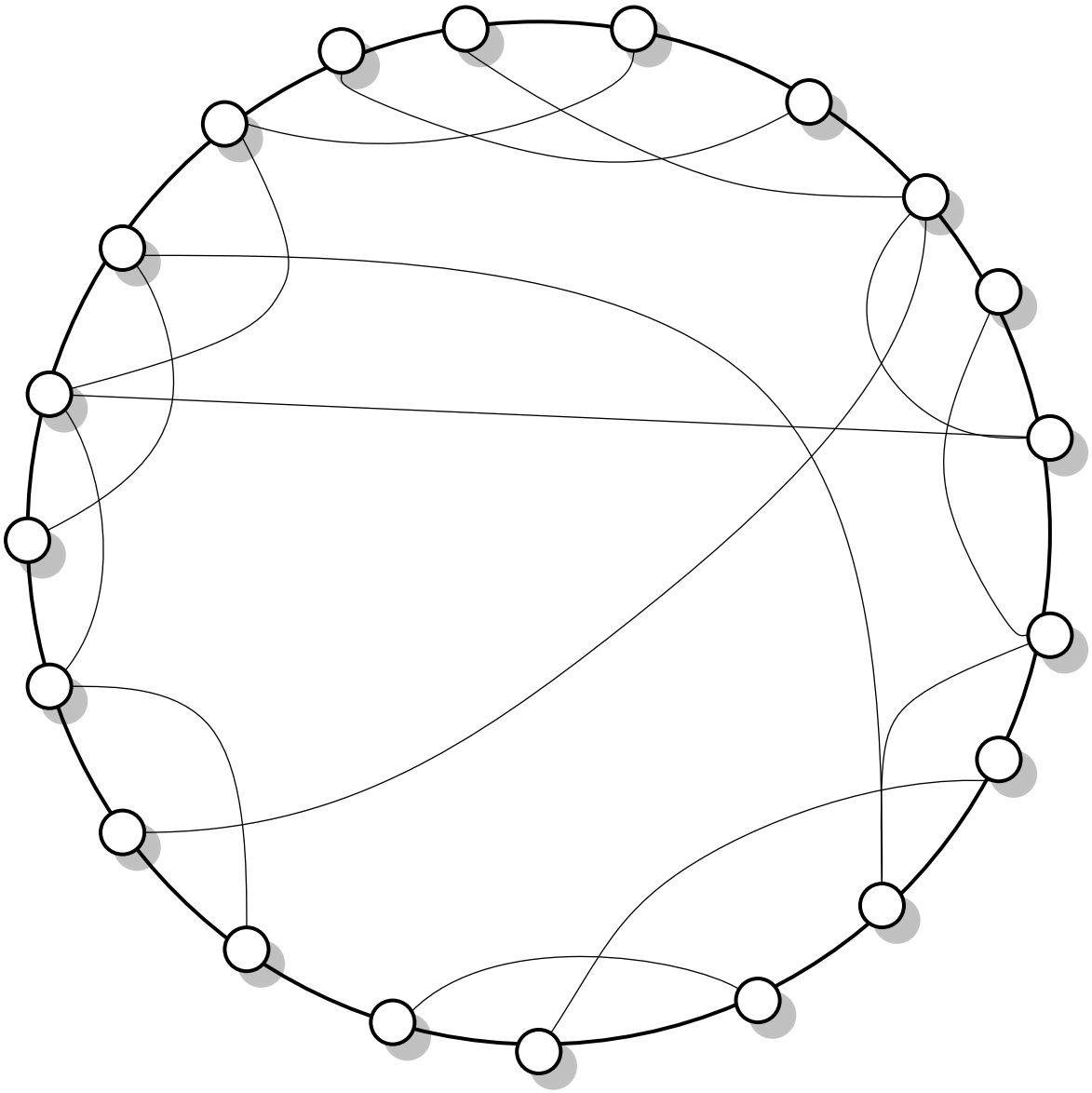


Figure 3.9: An example of a small-world network based on a cyclic list an randomly chosen long-range links

more practical networks which will also provide certain properties like a small diameter or short routing lengths. One class of more practical networks are small-world networks. We show that a self-stabilizing solution is not only possible for well-structured overlays but also for networks formed by a randomized process like a small-world network, in the sense that we still allow dynamics in the edges but show a convergence to and maintenance of desired network properties starting in any weakly connected state.

Small-world networks are named after the small-world phenomenon firstly observed by Milgrim [53] in 1967. In this work Milgrim observed the "six degrees of separation" in friendship graphs. The goal of the experiments was to determine whether the hypothesis that each pair of persons is connected by a short chain of acquaintances is true and if its true how many acquaintances such a chain consists of. A typical experiment worked as follows. A letter was sent to a person in Omaha, Nebraska - the source. The letter contained the name and address of a person in Boston, Massachusetts - the target. Now the person receiving the letter had to send it to the target directly if she knows this person by forename or send it to a person that is most likely to know the target. It turned out that in case of a successful delivery the chain had a length ranging from two up to ten with an average of five acquaintances.

From this empirical validation of the small-world phenomenon arose the question how a network with this property can be modeled. Even before Milgrim's experiments Kochen worked on a mathematical model for friendship graphs that could explain the small-world phenomenon that was published later [42]. Basically they explained the short lengths of chains of acquaintances by the small diameter occurring in random graphs. However they also pointed out in a realistic model acquaintances should not be selected randomly as it is more likely that a friend of person A knows another of A's friends and proposed different approaches to this problem. In networks modeling friendships, the probability of befriending a particular person is assumed to be inversely proportional to the number of geographically closer people, which matches the experimentally observed small-world property [53, 47, 18].

It turned out that the small-world property can not only be found in friendship graphs but also other real-world networks like the power grids, neural networks, social graphs [79]. Thus the research of small-world networks became popular again in the last years, especially after Watts and Strogatz [79] introduced their small-world network model. It is known that by taking a connected graph or network with a high graph diameter and adding a very small number of edges randomly, the diameter tends to drop drastically. In such a way networks can be constructed which have the small world property. The mechanism Watts and Strogatz introduced is a small-world network model that is capable of interpolating between a regular network and a random network using a single parameter. These graphs are constructed using a regular lattice, and on this lattice a process of constructing certain edges and probabilistic rewiring takes place, such that in the end a graph is formed that exhibits the small world properties. Watts and Strogatz argued that such a model captures two crucial parameters of social networks: there is a simple underlying structure that explains the presence of most edges, but a few edges are produced by a random process that does not respect this structure. They showed that a number of naturally arising networks exhibit this pair of properties. They also showed that the resulting graphs provide a high clustering coefficient even if only a few edges are affected by the random process. Thus small-world networks in the Watts-Strogatz model provide a certain robustness against node or link failures.

Kleinberg [34] showed based on the Watts-Strogatz model a mechanism not only constructing a small-world network, but also allowing a distributed algorithm to find the shortest paths in this network. In particular Kleinberg showed that greedy routing performs polylogarithmically using a k-harmonic distribution [34] in a k-dimensional lattice.

The variation of the small-world network we use in this work is that presented in [14]. In this work a distributed and randomized process is described that leads to a k-harmonic distribution of the long-range

links obtaining the same polylogarithmic performance like Kleinberg [34]. In this case, a graph G, given by a k-dimensional lattice, is enhanced with additional links chosen at random. More precisely, every node is given some long-range link pointing at another node in the graph. It turns out that for each long-range link added at a node u, the probability that the endpoint of this link is v is inversely proportional to the size of the ball of radius $dist(u, v)$ centered at u in G. So, in the k-dimensional lattice $\mathbb{Z}^k$, the probability that u has a long-range link pointing at v is essentially proportional to $1/d^k$ where d is the distance between u and v in the lattice. This setting of the long-range links enables greedy routing to perform in polylogarithmic expected number of steps (as a function of the distance in the lattice between the source and the target). In our work, we will focus on the 1-dimensional variation of the graph presented in [14]. We will try to form a self-stabilizing ring, which in addition contains long-range links as described above, giving us all the desired properties.

3.3.2 Move-Forget Process

We now give a more detailed description of the Move and Forget (M &F) Process established in [14]. As we mentioned above, each node establishes some links to other nodes. There exist two kinds of links, the links to the next known neighbors of the node and the ring edges that form the underlying ring network, as well as one long-range link for each node. The process that this long-range link follows is based on the Move-and-Forget Rewiring Process, described in [14]. The procedure works as follows: Assume a k-dimensional lattice $\mathbb{Z}^k$. In this lattice each node is initially occupied by exactly one token. These tokens move mutually independently according to random walks, starting from the node itself. That is, each token decides at each step its next position by altering its position in the lattice by ± 1 in each dimension with probability $\frac{1}{2}$. Nodes may forget their contacts through their long-range links. A long-range link of age $p.age = \alpha \geq 0$ (meaning that it is in existence for α steps) is forgotten with a probability $\phi(\alpha)$, where

$$\phi(\alpha) = \begin{cases} 0 & \text{if } \alpha = 0, 1 \text{ or } 2 \\ 1 - \frac{\alpha-1}{\alpha}\left(\frac{\ln(\alpha-1)}{\ln(\alpha)}\right)^{1+\epsilon} & \text{if } \alpha \geq 3 \end{cases}$$

ϵ is a fixed (arbitrary small) parameter of the algorithm. Note that the probability $\phi(\alpha)$ is independent of the number of dimensions k. When a long-range link is forgotten, the link stops existing and the token starts its random walk from the original node again. An illustration of the Move-& Forget process is given in Figure 3.10.

3.3.3 Our Contribution

In our work, we focus on the 1-dimensional variation of the graph presented in [14]. We propose a distributed self-stabilizing protocol that forms a ring which, in addition, contains long-range links as described above, giving us all the desired properties. In particular, the graph built by our protocol converges to a small-world network and, once established, the small-world properties are maintained. We present a protocol $P^{SMALL-WORLD}$ that solves the overlay problem $SMALL - WORLD$. We show the correctness of $P^{SMALL-WORLD}$ in the ATSS model in Theorem 3.3.1. The ideas of the protocol $P^{SMALL-WORLD}$ and the proof of correctness and the proof for the case of single joining or leaving nodes in the STSS model given in Theorem 3.3.15 have already been published in [36] which is a joint work with my colleague Andreas Koutsopoulos and our supervisor Christian Scheideler:

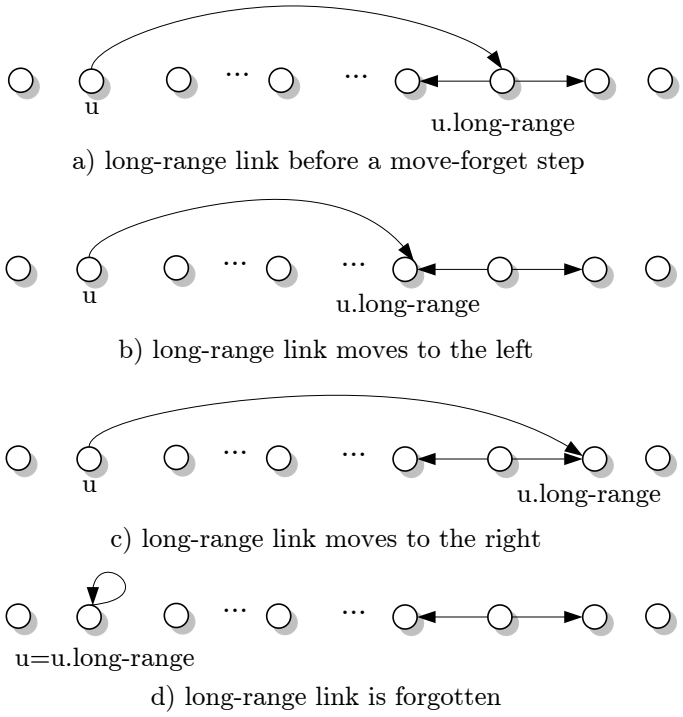


Figure 3.10: Three possible outcomes of a move-forget step

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler: A Self-Stabilization Process for Small-World Networks. IPDPS 2012

The results for the stabilization time in Theorem 3.3.8, stabilization work in Theorem 3.3.13 and maintenance work in Theorem 3.3.14 of the optimized protocol $P^{SMALL-WORLDsync}$ are presented for the first time.

3.3.4 Formal Definition

According to our model presented in Chapter 2 we assume that each node can be identified by an unique id. We define a hash function $h : ID \mapsto [0, 1]$ that maps the identifier of a node to a point in the $[0, 1]$ interval. We refer to $h(v)$ as v's position. We assume that h is chosen, such that no two nodes get the same position and that h is known to every node, i.e. by knowing the identifier of one node each node can determine its position. Then a global order on the nodes can be defined by their positions $h(v_1) < h(v_2) \cdots < h(v_n)$.

Definition 3.3.1 *A graph $G = (V, E)$ is a one dimensional small world network if:*

- $V = v_0, \cdots v_{n-1}$ *with* $h(v_i) < h(v_i + 1) \forall i \in \{0, \cdots n - 1\}$
- $E = \{(x_i, x_j) : j = i + 1 \mod n \vee j = i - 1 \mod n \vee j = k\}$ *such that the probability distribution of the number of nodes between x_i and x_k $|i - k|$ resembles the 1-harmonic distribution, i.e. probability $prob(|i - k| = d) \to \frac{1}{d}$.*

We now define the internal variables of each node used in our protocol to solve the overlay problem $SMALL-WORLD$. Each node u stores the following variables contributing to the explicit edge set:

- *u.predecessor*: The identifier of u's left neighbor ($h(u.predecessor) < h(u)$) or nil if there is no left neighbor.
- *u.successor*: The identifier of u's right neighbor ($h(u) < h(u.successor)$) or nil if there is no right neighbor.
- *u.long-range* The identifier of the node that u's long range link points to.
- *u.cycle*: The identifier of the node that u's cycle edge points to. Note that this identifier is only set if $u.successor = nil$ or $u.predecessor = nil$.

Additionally each node stores the following variables not contributing to the set of explicit edges.

- *u.Ch*: The channel for incoming messages.
- *u.age*: The age of the long range given by the number of rounds since the last reset of $u.long-range$.
- *u.*τ: a boolean variable that is periodically true. Note that we don't allow the adversary to alter $u.age$.

To communicate with each other, nodes send and receive messages. The content of a message is given by the ids of some nodes and a specification of the message type. So different types of messages can trigger different actions. Thus a message m is of the following form $m = (type, ids)$.

- *forward*: The standard message type to forward edges.
- *long-range*: This message type is used to mark incoming long range links that form the small-world network.
- *move-forget*: This message is sent to respond to an incoming long range link and to inform the origin of the long range link about possible network changes.
- *cycle*: This type of message is used to establish a cycle edge if a node misses its left neighbor.
- *response-cycle*: This message is sent to respond to an incoming cycle edge and to inform the source of the cycle link about possible network changes.
- *probing-right*: This type of message is used to propagate the probing message to the right.
- *probing-left*: This type of message is used to propagate the probing message to the left.

In the following we define when an assignment of the variables is valid and how the sets of initial topologies IT and the goal topologies $SMALL-WORLD$ look like.

Definition 3.3.2 *An assignment of the variables of a node is* valid *if* $h(u.predecessor) < h(u)$ *or* $u.predecessor = nil$ *and* $h(u.successor) > h(u)$ *or* $u.successor = nil$. *Furthermore* $u.cycle \neq nil$ *only if* $u.successor = nil$ *or* $u.predecessor = nil$ *and if* $h(u.long - range) > h(u)$, *then* $h(u.long - range) > h(u.successor)$ *and if* $h(u.long - range) < h(u)$, *then* $h(u.long - range) < h(u.predecessor)$. *Note that an invalid assignment can be locally repaired immediately. Thus we assume in the following w.o.l.g. that initially the assignment is valid for every node.*

With the given internal variables we are now able to define the set of initial topologies IT and especially the set of goal topologies $SMALL - WORLD$.

Definition 3.3.3 *Let* E_e *and* E_i *be defined as described in Chapter 2 according to the definition of internal variables. Then the* set of initial topologies *is given by:*

$$IT = \{G = (V, E = E_e \cup E_i) : G \text{ is weakly connected}\}$$

Definition 3.3.4 *Let* E_e *and* E_i *be defined as described in Chapter 2 according to the definition of internal variables. Then the set of* target topologies *is given by:*

$$SMALL - WORLD = \{G = (V, E) : G_e \text{ is a 1-dimensional small-world network}\}$$

3.3.5 Protocol $P^{\text{SMALL-WORLD}}$

The concept of our protocol for a self-stabilizing 1-dimensional small-world network consists of three parts. At first we reuse the protocol P^{LIST} to form a sorted list. We then introduce further actions to establish an edge between the nodes with the maximal and minimal position to create a closed cyclic list. This cyclic list is the same as a 1-dimensional lattice. By further actions the Move & Forget-process is implemented. Then eventually a 1-dimensional small-world network is formed.

The basic approach to form a sorted list as presented in Section 3.2 is extended by using the long-range links as shortcuts when u forwards a node v if $h(v) > h(u.long - range) > h(u.successor)$ (resp. $h(v) < h(u.long - range) < h(u.predecessor)$). We therefore have to ensure that the sorted list is established between u and $u.long - range$. To ensure this connectivity we will introduce a technique called probing which we discuss separately.

As the Move & Forget-process requires not only a sorted list as a structure but a cyclic list we need further messages and actions. If a node u determines based on its local information that it is a node of maximal or minimal position, i.e. $u.successor = nil$ or $u.predecessor = nil$, then u stores the node of minimal (resp. maximal) position it is aware of in $u.cycle$. Eventually $u.cycle$ contains either the node of minimal (resp. maximal) position v_0 (resp. v_{n-1}) in the network or u is informed about a node v with $h(v) > h(u)$ (resp. $h(v) < h(u)$), such that $u.successor \neq nil$ (resp. $u.predecessor \neq nil$). In the end this leads to an edge between v_0 and v_{n-1} and vice versa, i.e. $v_0.cycle = v_{n-1}$ and $v_{n-1}.cycle = v_0$. Obviously local information does not suffice to find the node of minimal or maximal position in the network. Therefore we additionally need the local knowledge of our neighbors. In fact we send a message to the node currently stored in $u.cycle$ as the node of minimal (resp. maximal) position u is aware of and trigger a response message containing the node with minimal (resp. maximal) position $u.cycle$ is aware of, so u can update its internal variable.

As a last step in our protocol we implement the M & F process for a 1-dimensional lattice. For each node u its long-range link $u.long - range$ is moved to the successor or predecessor of the current

$u.long - range$. As there are two possibilities to move the endpoint of the long-range link each one is chosen with probability $\frac{1}{2}$. This is achieved in the following way: whenever τ is true u sends a message to $u.long - range$ of type long-range and $u.long - range$ responds by sending the identifiers of its successor and predecessor. Then u chooses one as its new $u.long - range$. Additionally u forgets the long-range link with probability $\phi(u.age)$.

About Probing

A general problem in topological self-stabilization is the restriction of locality, i.e. each node only knows its direct neighbors stored in internal variables. By local knowledge alone failures in the topology can not always be detected. Two examples are already mentioned above. How can the protocol ensure that u and $u.long - range$ are connected in the sorted list? How can u be sure that $u.cycle$ contains the node with the minimal or maximal position? Both questions can't be answered by u itself based only on u's local knowledge. One can think of several solutions of this problem. One possible solution is to extend the locality of each node, i.e. a node does not only know its own neighbor but also its neighbor's neighbors. A problem with this model is that if one wants to implement it the nodes have to inform all their neighbors about their current neighborhood. This leads to a high amount of work in terms of sent and received messages. In fact the work is then up to $\mathcal{O}(n^2)$ for one node in one round.

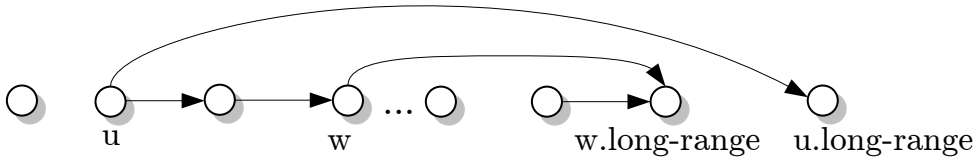
We already sketched another solution by discussing how to form a cyclic list out of the sorted list. Instead of extending the locality we can also extend the neighborhood of the nodes. So additionally to the edges needed by the target topology each node maintains further explicit or implicit edges, such that the node is enabled to check properties that need more than local knowledge. We call this technique *probing messages*.

In the case of small-world networks we show how connectivity can be ensured by using probing messages. In particular we want to be sure that nodes u and $u.long - range$ are connected in the sorted list, i.e. there is a path from u to $u.long-range$ following only edges to successors or predecessors. There is the trivial approach to ensure this connectivity by simply forwarding $u.long - range$ to $u.successor$ or $u.predecessor$ by a forward message in each round. As there can be up to $n-2$ nodes in the sorted list between u and $u.long - range$ this means that there is a high maintenance work of $\mathcal{O}(n)$ implicit edges (messages) for each pair $u, u.long - range$ in each round to ensure this connectivity. We can decrease this amount by introducing specific probing messages of type $probing-right$ and $probing-left$. These messages not necessarily forwarded to the successor or predecessor of the receiving node but are routed according to a greedy routing scheme to the neighbor of which the position is closest to the position of the node contained in the probing message, i.e. we use possible long-range links as short-cuts. As we show in the analysis this reduces the maintenance work. If the probing message fails to reach $u.long - range$, i.e. it can not be forwarded anymore then the receiving node v creates a link to $u.long - range$ by storing it as its successor or predecessor. An example of the probing process is illustrated in Figure 3.11.

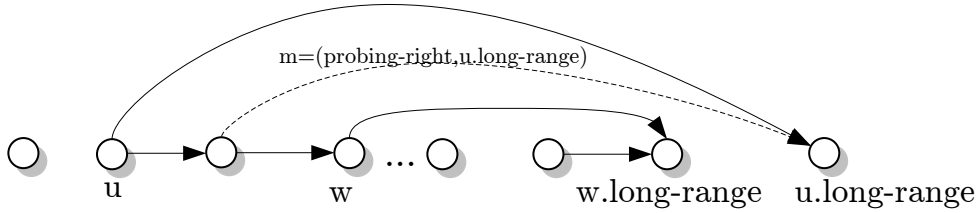
We present a pseudo code implementation of $P^{SMALL-WORLD}$ including a probing mechanism in Algorithms 3.3.1-3.3.10.

3.3.6 Analysis in the ATSS model

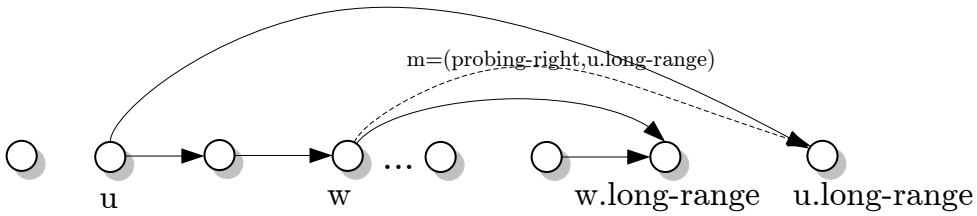
We show the correctness of $P^{SMALL-WORLD}$ in the ATSS model of our approach by proving that each initially weakly connected graph stabilizes to a small-world network. We show this by dividing the self-stabilization process into different phases and determine the correctness of each phase. In particular we will show that if each node executes $P^{SMALL-WORLD}$ started in a weakly connected graph, eventually



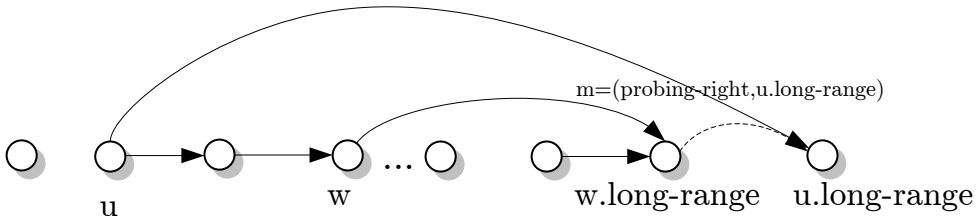
a) u starts probing to u.long-range



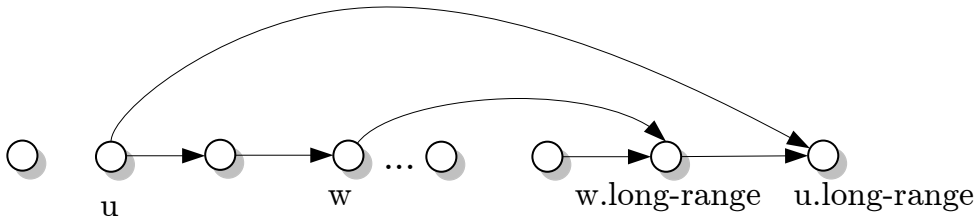
b) u send probing message to its successor



c) probing message is forwarded to w



d) w forwards probing message to w.long-range



e) w.long-range creates explicit edge to w.long-range

Figure 3.11: An example of a probing process for u and $u.long-range$

Algorithm 3.3.1 $P^{SMALL-WORLD}$

message $m = (type, v) \in u.Ch \rightarrow$
if type=forward **then**
    forward(v)
else if type=long-range **then**
    long-range(v)
else if type=probing-right **then**
    probing-right(v)
else if type=probing-left **then**
    probing-left(v)
else if type=cycle **then**
    cycle(v)
else if type=response-cycle **then**
    response-cycle(v)
message $m = (type, v_0, v_1, v_2) \in u.Ch \rightarrow$
if type=move-forget **then**
    move-forget(v_0, v_1, v_2)
$\tau \rightarrow$
introduction()
probing()

a graph is reached, that contains the sorted list as a sub-graph. We then show that if started in a graph that contains the sorted list as a sub-graph $P^{SMALL-WORLD}$ leads to a graph containing the cyclic list as a sub-graph. We proceed by showing that started in a graph that contains the cyclic list a a sub-graph eventually a small-world network is reached. Thus we can show the main theorem:

Theorem 3.3.1 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLD}$ then eventually the graph converges to a graph $G' \in SMALL - WORLD$ (Convergence). If in an initial graph $G \in SMALL - WORLD$ every node executes the protocol $P^{SMALL-WORLD}$ then each possible computation leads to a graph $G' \in SMALL - WORLD$ (Closure).*

To prove this theorem we have to introduce some formal definition of the intermediate graphs and some subset of the edges.

Definition 3.3.5 *We define:*

- $E_{e-list} = \{(u, v) \in E_e : v = u.predecessor \vee v = u.successor\} \subseteq E_e$ *is the subset of explicit edges that will form the sorted list.*
- $E_{i-list} = \{(u, v) \in E_i : \exists m = (forward, v) \in u.Ch\}$ *is the subset of implicit edges that are needed to form the sorted list.*
- $E_{list} = E_{e-list} \cup E_{i-list}$ *is the set of edges that take part in the process of forming the sorted list.*

Algorithm 3.3.2 FORWARD(v)

if $h(v) > h(u)$ **then**
  if $u.successor \neq nil$ **then**
    if $h(v) < h(u.successor)$ **then**
      send message m=(forward,$u.successor$) to v
      $u.successor = v$
    else if $h(v) > h(u.long - range) > h(u.successor)$ **then**
      send message $m = (forward, v)$ to $u.long - range$
    else
      send message $m = (forward, v)$ to $u.successor$
  else
    $u.successor = v$
    if $u.cycle \neq nil$ **then**
      forward($u.cycle$)
      $u.cycle = nil$
else if $h(v) < h(u)$ **then**
  if $u.predecessor \neq nil$ **then**
    if $h(v) > h(u.predecessor)$ **then**
      send message $m = (forward, u.predecessor)$ to v
      $u.successor = v$
    else if $h(v) < h(u.long - range) < h(u.predecessor)$ **then**
      send message $m = (forward, v)$ to $u.long - range$
    else
      send message $m = (forward, v)$ to $u.successor$
  else
    $u.predecessor = v$
    if $u.cycle \neq nil$ **then**
      forward($u.cycle$)
      $u.cycle = nil$

Algorithm 3.3.3 LONG-RANGE(V)

if $u.predecessor \neq nil \wedge u.successor \neq nil$ **then**
  send message $m = (respond - long - range, u.predecessor, u.successor, u)$ to v
else if $u.predecessor \neq nil$ **then**
  send message $m = (respond - long - range, u.predecessor, u.cycle, u)$ to v
else if $u.successor \neq nil$ **then**
  send message $m = (respond - long - range, u.cycle, u.successor, u)$ to v
elseforward(v)

Algorithm 3.3.4 PROBING-RIGHT(v)

if $h(v) \geq h(u.long - range) > h(u.successor)$ **then**
  send message $m = (probing - right, v)$ to $u.long - range$
else if $h(v) \geq h(u.successor) \wedge u.successor \neq nil$ **then**
  send message $m(probing - right, v)$ to $u.successor$
else
  forward(v)

Algorithm 3.3.5 PROBING-LEFT(v)

if $h(v) \leq h(u.long - range) < h(u.predecessor)$ **then**
  send message $m = (probing - left, v)$ to $u.long - range$
else if $h(v) \leq h(u.predecessor) \wedge u.predecessor \neq nil$ **then**
  send message $m = (probing - left, v)$ to $u.predecessor$
else
  forward(v)

Algorithm 3.3.6 CYCLE(v)

if $h(v) < h(u)$ **then**
  if $h(u.long - range) < h(v)$ **then**
    send message $m = (forward, u.long - range)$ to v
  else if $h(u.predecessor) < h(v) \wedge u.predecessor \neq nil$ **then**
    send message $m = (forward, u.predecessor)$ to v
  else if $h(u.long - range) > h(v)$ **then**
    send message $m = (response - cycle, u.long - range)$ to v
  else if $u.successor \neq nil$ **then**
    send message $m = (response - cycle, u.successor)$ to v
  else
    send message $m = (response - cycle, u)$ to v
else
  if $h(u.long - range) > h(v)$ **then**
    send message $m = (forward, u.long - range)$ to v
  else if $h(u.successor) > h(v) \wedge u.successor \neq nil$ **then**
    send message $m = (forward, u.successor)$ to v
  else if $h(u.long - range) < h(u.predecessor)$ **then**
    send message $m = (response - cycle, u.long - range)$ to v
  else if $u.predecessor \neq nil$ **then**
    send message $m = (response - cycle, u.predecessor)$ to v
  else
    send message $m = (response - cycle, u)$ to v

Algorithm 3.3.7 RESPONSE-CYCLE(v)

if $u.predecessor = nil \wedge (h(v) > h(u.cycle) > h(u) \vee (u.cycle = nil \wedge h(v) > h(u)))$ **then**
  if $u.cycle \neq nil$ **then**
    forward($u.cycle$)
  $u.cycle = v$
else if $u.successor = nil \wedge (h(v) < h(u.cycle) < h(u) \vee (u.cycle = nil \wedge h(v) < h(u)))$ **then**
  if $u.cycle \neq nil$ **then**
    forward($u.cycle$)
  $u.cycle = v$
elseforward(v)

Algorithm 3.3.8 MOVE-FORGET(v_0, v_1, v_2)

if $v_2 = u.long - range$ **then**
  Choose $i \in 0, 1$ with probability $\frac{1}{2}$ each
  send message $m = (forward, u.long - range)$ to v_i
  send message $m = (forward, v_{1-i})$ to v_i
  $u.long - range = v_i$
  $reset = true$ with probability $\phi(u.age)$
  if reset=true **then**
    forward($u.long - range$)
    $u.long - range = u$
  else
    probing($u.long - range$)
else
  forward(v_0),forward(v_1),forward(v_2)

Algorithm 3.3.9 INTRODUCTION()

if $h(u.predecessor) \neq nil$ **then**
  send message $m = (forward, u)$ to $u.predecessor$
else
  send message $m = (cycle, u)$ to $u.cycle$
if $h(u.successor) \neq nil$ **then**
  send message $m = (forward, u)$ to $u.successor$
else
  send message $m = (cycle, u)$ to $u.cycle$
send message $m = (long - range, u)$ to $u.long - range$

Algorithm 3.3.10 PROBING()

if $h(u.long - range) < h(u)$ **then**
  if $h(u.long - range) \leq h(u.predecessor)$ **then**
    send message $m = (probing - left, u.long - range)$ to $u.predecessor$
else
  if $u.long - range \geq u.successor$ **then**
    send message $m = (probing - right, u.long - range)$ to $u.successor$
if $u.cycle \neq nil \wedge h(u.cycle) < h(u.predecessor)$ **then**
  send message $m = (probing - left, u.cycle)$ to $u.predecessor$
else if $u.cycle \neq nil \wedge h(u.cycle) > h(u.successor)$ **then**
  send message $m = (probing - right, u.cycle)$ to $u.successor$

- $E_{e-cycle} = \{(u, v) \in E_e : v = u.cycle\} \cup E_{e-list} \subseteq E_e$ *is the subset of explicit edges that will form the cyclic list.*
- $E_{i-cycle} = \{(u, v) \in E_i : \exists m = (cycle, v) \vee m = (response - cycle, v) \in u.Ch\} \cup E_{i-list}$ *is the subset of implicit edges that are needed to form the cyclic list.*
- $E_{cycle} = E_{e-cycle} \cup E_{i-cycle}$ *is the set of edges that take part in the process forming the cyclic list.*

Definition 3.3.6 *A graph $G = (V, E)$ is a* sorted list*, if $V = v_0, \cdots v_{n-1}$ with $h(v_i) < h(v_i + 1) \forall i \in \{0, \cdots n - 1\}$ and $E = \{(v_i, v_j) :, j = i - 1 \vee j = i + 1 \forall 0 < j < n - 1\}$.*

The definition of the sorted list is analogous to the definition used in Section 3.2, in which we solved the sorted list overlay problem.

Definition 3.3.7 *A graph $G = (V, E)$ is a* cyclic list*, if $V = v_0, \cdots v_{n-1}$ with $h(v_i) < h(v_{i+1}) \forall i \in \{0, \cdots n - 1\}$ and $E = \{(v_i, v_{i+1 \mod n}), (v_{+1}, v_{i \mod n})\}$.*

An illustration of a cyclic list is given in Figure 3.12.

Convergence

We start by showing that the first part of theorem 3.3.1 holds and $P^{SMALL-WORLD}$ fulfills the convergence property.

Theorem 3.3.2 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLD}$ then eventually the graph converges to a graph $G' \in SMALL - WORLD$ (Convergence).*

To show this we prove a set of intermediate theorems first, from which the convergence follows. Our first theorem claims that the computation reaches a state, such that the sorted list is a sub-graph of the graph of explicit edges.

Theorem 3.3.3 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLD}$ then eventually the computation reaches a state s_t such that G^t_{e-list} is a sorted list.*

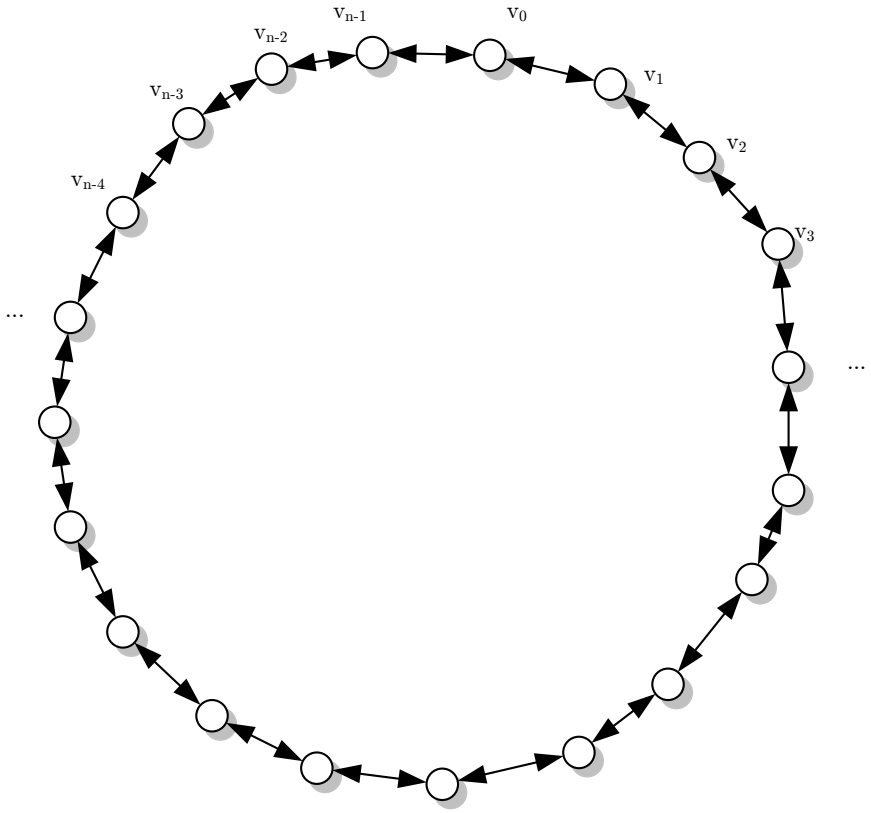


Figure 3.12: An example of a cyclic list

To prove this theorem we first show a helpful lemma.

Lemma 3.3.1 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLD}$ then eventually the computation reaches a state s_t such that G^t_{list} is weakly connected and stays connected in every state $s_{t'}$, $t' > t$.*

Proof. We start the proof by showing that if G^t is weakly connected also G^{t+1} stays weakly connected. We therefore assume that an edge $(u, v) \in E^t$ existing at time t does not exist at time $t + 1$ and show that u and v stay connected. We consider any possible explicit and implicit edge.

If $(u, v) \in E^t_e$ but $(u, v) \notin E^{t+1}$ and

- $v = u.successor$ at time t, then u forwards v to its new successor w at time $t+1$ and $(u, w) \in E^{t+1}_e$ and $(w, v) \in E^{t+1}_i$. Thus u and v stay connected.
- $v = u.predecessor$ at time t, then by symmetric arguments u and v stay connected.
- $v = u.cycle$ at time t, then u received a new node w with $h(w) < h(v) < h(u)$ (resp. $h(w) > h(v) > h(u)$) and forwards v, thus by the same arguments as above u and v stay connected.
- $v = u.long - range$, then u received a move-forget message and either forwards u to the new endpoint w of the long-range link or forwards v to its successor w or predecessor w if u forgets the long-range link. Then $(u, w) \in E^{t+1}_e$ and $(w, v) \in E^{t+1}_i$.

If $(u, v) \in E^t_i$ but $(u, v) \notin E^{t+1}$ and

- v is stored in a forward message, then u receives this message and forwards v to $w = u.successor$ or $w = u.predecessor$. Then $(u, w) \in E^{t+1}_e$ and $(w, v) \in E^{t+1}_i$ and u and v stay connected.
- v is stored in an long-range message, then u sends a message containing at least one of $u.successor$, $u.predecessor$ and $u.cycle$ to v w.l.o.g. we assume that the message contains $w = u.successor$, then $(u, w) \in E^{t+1}_e$ and $(v, w) \in E^{t+1}_i$, or u forwards v in a forward message. Then u and v stay connected.
- v is stored in a cycle message, then u responds to v by a message containing at least one of $u.successor$, $u.predecessor$ and $u.long - range$ or u itself to v. W.l.o.g. we assume that the message contains $w = u.successor$, then $(u, w) \in E^{t+1}_e$ and $(v, w) \in E^{t+1}_i$ and u and v stay connected.
- v is stored in a response-cycle message, then either v is forwarded by a forward message or v becomes the new $u.cycle$. Either way u and v stay connected.
- v is stored in a probing-right message, then u forwards v to $u.successor$ or $u.long - range$ or stores v as its new $u.successor$. In both cases u and v stay connected.
- v is stored in a probing-left message, then u forwards v to $u.predecessor$ or $u.long - range$ or stores v as its new $u.predecessor$. In both cases u and v stay connected.
- v is stored in a move-forget message, then either v is forwarded to the new endpoint of the long-range link or v is the new $u.long - range$. Either way u and v stay connected.

After establishing that connectivity is preserved, we define a time t_0 at which every message initially in $u.Ch$ for any node u and also the messages in response have been received. Then we know that for every long-range link $u.long - range$ at time $t_i \geq t_0$ u started a probing. Based on this fact we can show that eventually a state s_j at time $t_j \geq t_0$ in the computation is reached such that $G_{list}^{t_j}$ is weakly connected and stays weakly connected in any state s_k at time $t_k \geq t_j$.To show that $G_{list}^{t_j}$ stays connected we consider the cases in which the connectivity in G_{list}^{t} might get lost.

If $(u, v) \in E_{list}^{t}$, then either $v = u.successor$ or $v = u.predecessor$ or $m = (forward, v) \in u.Ch$. In the first case connectivity is maintained, as u either keeps the edge or receives a node w as its new successor and sends a message $m = (forward, v)$ to w. Then $(u, w) \in E_{e-list}^{t+1}$ and $(w, v) \in E_{i-list}^{t+1}$. In the second case the connectivity is maintained if u stores v as its new successor or predecessor or if v is forwarded by a message $m = (forward, v)$ to $u.successor$ or $u.predecessor$. Only if u forwards v to $u.long - range$ with $h(u) < h(u.long - range) < h(v)$ (resp. $h(u) > h(u.long - range) > h(v)$) connectivity in G_{list}^{t+1} can be lost between u and v. We show that the connectivity eventually is restored and only lost again a finite number of times, so eventually u and v stay connected in $G_{list}^{t_j}$ at some time $t_j \geq t_0$.

W.l.o.g. we assume $h(v) < h(u)$ in the following. We now consider an edge $(u, v) \in E_{list}^{t_0}$. Either u and v stay connected in every state afterwards, so for this edge the lemma already holds or as we have observed v is forwarded using a long-range link. We denote the time at which v is forwarded by u by t_1 then the edge (u, v) is replaced by a path $(u, w_1), (w_1, v) \in E^{t_1+1}$ where $w_1 = u.successor$ or $w_1 = u.long - range$ at time t_1 and $h(u) < h(w_1) < h(v)$.

Let w_i be the node v is forwarded to at time t_i. Now again either w_i and v stay connected at every state afterwards or v is forwarded to a node w_{i+1} at some time t_{i+1}. Then for w_i and w_{i+1} it holds that $h(u) < h(w_i) < h(w_{i+1}) < h(v)$. Then the number of times v is forwarded is bounded by $\mathcal{O}(n)$ and eventually v can not be forwarded anymore. This forwarding procedure for v started at u gives us a sequence of nodes $fs(u, v) = (u = w_0, w_1, w_2, \cdots w_j, w_{j+1} = v)$. We call $fs(u, v)$ a forwarding sequence for u and v. In $fs(u, v)$ w_j stays connected with v in every state s_k at time $t_k > t_j$ and $w_i = w_{i-1}.successor$ or $w_i = w_{i-1}.long - range$ at time t_i.

If $w_i = w_{i-1}.successor$ then again either w_{i-1} and w_i stay connected in every state afterwards t_i or w_{i-1} forwards w_i. Thus by the same arguments as for (u, v) we can replace (w_{i-1}, w_i) by a forwarding sequence $fs(w_{i-1}, w_i) = (w_{i-1} = w_{(i-1,i),0}, w_{(i-1,i),1}, \cdots w_{(i-1,i),l}, w_{(i-1,i),l+1} = w_i)$ with $h(w_{i-1}) < h(w_{(i-1,i),j-1}) < h(w_{(i-1,i),j}) < h(w_i)$ and $w_{(i-1,i),j} = w_{(i-1,i),j-1}.successor$ or $w_{(i-1,i),j} = w_{(i-1,i),j-1}.long - range$.

If $w_i = w_{i-1}.long - range$ then according to our observation w_i sends or has sent a probing message containing w_i. A probing message $m = (probing - right, v) \in u.Ch$ is only forwarded from u to $u.successor$ or $u.long - range$ whereby $h(u) < h(u.successor) < h(v)$ or $h(u) < h(u.long - range) < h(v)$. If v can not be forwarded an edge $(u, v) \in E_{e-list}$ is created. Thus exactly like a forwarding sequence also a probing path gives us a sequence of nodes $ps(u, v) = (u = w_0, w_1, w_2, \cdots w_j, w_{j+1} = v)$. We call $ps(u, v)$ a probing sequence for u and v. In $ps(u, v)$ w_j stays connected with v in every state s_k at time $t_k > t_j$ and $w_i = w_{i-1}.successor$ or $w_i = w_{i-1}.long - range$ at time t_i.

Then we can replace (w_{i-1}, w_i) in $fs(u, v)$ by such a probing sequence $ps(w_{i-1}, w_i) = (w_{i-1} = w_{(i-1,i),0}, w_{(i-1,i),1}, \cdots w_{(i-1,i),l}, w_{(i-1,i),l+1} = w_i)$ with $h(w_{i-1}) < h(w_{(i-1,i),j-1}) < h(w_{(i-1,i),j}) < h(w_i)$ and $w_{(i-1,i),j} = w_{(i-1,i),j-1}.successor$ or $w_{(i-1,i),j} = w_{(i-1,i),j-1}.long - range$. An example of such a replacement is given in Figure 3.13.

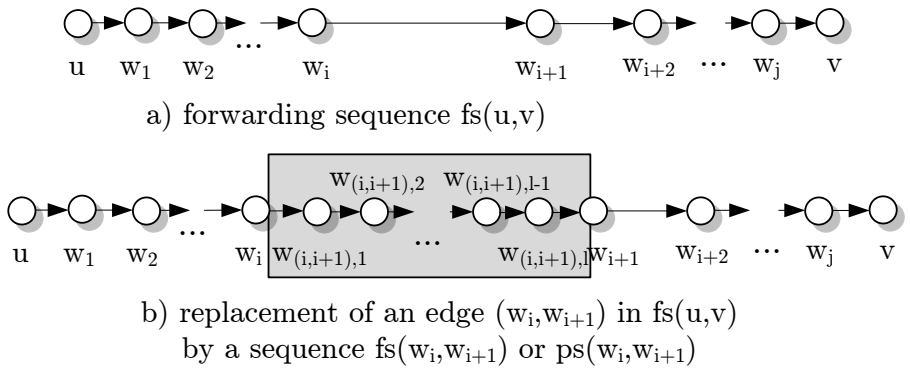


Figure 3.13: An example of replacements in a forwarding sequence $fs(u, v)$

Inductively we can apply the same arguments on the replacing of forwarding or probing sequences. As the number of nodes is finite, eventually at some time t' we get a connecting sequence $cs(u, v) = (u = \tilde{w}_0, \tilde{w}_1, \tilde{w}_2, \cdots \tilde{w}_l, \tilde{w}_{l+1} = v)$, where $\tilde{w}_{j-1}$ stays connected in G^t_{list} with $\tilde{w}_j$ in every state afterwards time $t \geq t'$ and $\tilde{w}_j = \tilde{w}_{j-1}.successor$ or $\tilde{w}_j = \tilde{w}_{j-1}.long - range$. We get $cs(u, v)$ starting from the forwarding sequence $fs(u, v)$ of v from u and replacing each edge in this sequence that is a long-range link or edge in E_{list} that gets disconnected by a probing sequence $ps(w_i, w_{i+1})$ or forwarding sequence $fs(w_i, w_{i+1})$.

The same arguments that we used to show that if an edge $(u, v) \in E^0_{list}$ gets disconnected in G^t_{list} at some time t then they are weakly connected in $G^{t'}_{list}$ and stay weakly connected in every state afterwards can be applied for any edge in E^0.

If $v = u.long - range$ we start with a probing sequence instead of a forwarding sequence and eventually no edges in this sequence can be replaced such that u and v are weakly connected in $G^{t'}_{list}$ and stay weakly connected in every state afterwards.

If $m = (probing - right, v) \in u.Ch$ (resp. $m = (probing - left, v) \in u.Ch$) again there is a probing sequence $ps(u, v)$ from u to v. Then with the same arguments as for $v = u.long - range$ eventually u and v are weakly connected in $G^{t'}_{list}$ and stay weakly connected in every state afterwards.

If $m = (move - forget, v_0, v_1, v_2) \in u.Ch$ there can be several cases. If $v_2 \neq u.long - range$, v_0, v_1, v_2 are treated as identifiers in forward messages, for which we already showed that eventually weak connectivity in $G^{t'}_{list}$ is ensured. If w.l.o.g. v_0 becomes the new long-range link, this can be reduced to the case $v = u.long - range$. Then v_1 and v_2 are forwarded to v_0 by forward messages and thus eventually u will stay weakly connected with v_1 and v_2 via the node v_0. If the long-range link is forgotten v_0 is treated as an identifier in a forward message and again u and v_0 will eventually stay weakly connected in $G^{t'}_{list}$.

If $v = u.cycle$ either u starts a probing to v and this can be reduced to the case $v = u.long - range$, or before u starts a probing to v $u.cycle$ is updated and v is treated as an identifier in a forward message.

If $m = (response - cycle, v) \in u.Ch$ either $v = u.cycle$ or v is treated as an identifier in a forward message. If $m = (long - range, v) \in u.Ch$ then u responds by sending a move-forget message containing u. Then this can be reduced to $m = (move - forget, v_0, v_1, u) \in v.Ch$. Or if u can not respond to v, v is treated as an identifier in a forward message.

In any case eventually there is a state $s_{t'}$ such that in any later state s_t, $t \geq t'$, u and v stay weakly connected in G^t_{list}. □

In the following we denote by t_0 the time such that all nodes are weakly connected in $G_l^t ist$ with $t \geq t_0$ and stay weakly connected. From Lemma 3.3.1 follows that the computation reaches such a state s_{t_0} Thus we can now show that $G_{list}^{t_0}$ eventually converges to a sorted list. To show this convergence we reuse the analysis presented in the proof of Theorem 3.2.2, where we showed the convergence property for P^{LIST}.

Lemma 3.3.2 *If the computation of $P^{SMALL-WORLD}$ reaches a state s_t at a time $t \geq t_0$ where for some node u there are two edges $(u, v) \in E_{e-list}^t$ and $(u, w) \in E_{i-list}^t$ such that $h(u) < h(w) < h(v)$ (resp. $h(v) < h(w) < h(u)$) then this computation contains a later state $s_{t'}$, $t' > t$, with an edge $(u, w') \in E_{e-list}^{t'}$ with $h(w') \leq h(w)$ (resp. $h(w') \leq h(w)$).*

Remember that this lemma claims that the stored edges $(u, u.successor)$ and $(u, u.predecessor)$ are shortened over time.

Proof. The proof is analogous to the proof of Lemma 3.2.2. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v)$ used in $P^{SMALL-WORLD}$ and the sets E_e^t by E_{e-list}^t, E_i^t by E_{i-list}^t and E^t by E_{list}^t. □

Lemma 3.3.3 *If the computation of $P^{SMALL-WORLD}$ reaches a state s_t, $t \geq t_0'$, where for a node u there are edges $(u, v) \in E_{e-list}^t$ and $(u, w) \in E_{i-list}^t$ with $h(u) < h(v) < h(w)$ (resp. $h(u) > h(v) > h(w)$) then the computation contains a later state $s_{t'}$, $t' > t$, where there is an edge $(w', w) \in E_{list}^{t'}$ with $h(u) < h(w') < h(w)$ (resp. $h(u) > h(w') > h(w)$).*

Proof. The proof is analogous to the proof of Lemma 3.2.3. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v)$ used in $P^{SMALL-WORLD}$ and the sets E_e^t by E_{e-list}^t, E_i^t by E_{i-list}^t and E^t by E_{list}^t. □

Lemma 3.3.4 *If the computation of $P^{SMALL-WORLD}$ reaches a state s_t, $t \geq t_0'$, where for some nodes u, v and w such that $h(u) < h(w) < h(v)$ (resp. $h(v) < h(w) < h(u)$) there are edges $(u, v) \in E_{e-list}^t$ and $(w, u) \in E_{e-list}^t$ then the computation contains a later state $s_{t'}$, $t' > t$, where either some edge in $E_{e-list}^{t'}$ is shorter than in E_{e-list}^t or $(u, w) \in E_{e-list}^{t'}$.*

Proof. The proof is analogous to the proof of Lemma 3.2.4. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v)$ used in $P^{SMALL-WORLD}$ and the sets E_e^t by E_{e-list}^t, E_i^t by E_{i-list}^t and E^t by E_{list}^t. □

Lemma 3.3.5 *If the computation of $P^{SMALL-WORLD}$ reaches a state s_t, $t \geq t_0'$, where there is an edge $(u, v) \in E_{e-list}^t$ then the computation contains a later state $s_{t'}$, $t' > t$, where some edge in $E_{e-list}^{t'}$ is shorter than in E_{e-list}^t or $(v, u) \in E_{e-list}^{t'}$.*

Proof. The proof is analogous to the proof of Lemma 3.2.5. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v)$ used in $P^{SMALL-WORLD}$ and the sets E_e^t by E_{e-list}^t, E_i^t by E_{i-list}^t and E^t by E_{list}^t. □

Lemma 3.3.6 *If the computation of $P^{SMALL-WORLD}$ reaches a state s_t, $t \geq t_0'$, such that $(u, v) \in E_{e-list}^{t'} \Rightarrow (v, u) \in E_{e-list}^{t'}$ in every state $s_{t'}$, $t' > t$, after, then this computation contains a state s_{t*} such that E_{e-list}^{t*} is strongly connected.*

Proof. The proof is analogous to the proof of Lemma 3.2.6. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v)$ used in $P^{SMALL-WORLD}$ and the sets E_e^t by E_{e-list}^t, E_i^t by E_{i-list}^t and E^t by E_{list}^t. □

Lemma 3.3.7 *If the computation of $P^{SMALL-WORLD}$ reaches a state s_t, $t \geq t_0'$, such that E_{e-list}^t is strongly connected and for every pair of nodes $(u, v) \in E_{e-list}^t \Rightarrow (v, u) \in E_{e-list}^t$ then this state is a solution for the sorted list problem and $G_l^t ist$ is a sorted list.*

Proof. The proof is analogous to the proof of Lemma 3.2.7. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v)$ used in $P^{SMALL-WORLD}$ and the sets E_e^t by E_{e-list}^t, E_i^t by E_{i-list}^t and E^t by E_{list}^t. □

We are now ready to show Theorem 3.3.3. According to Lemma 3.3.1 G_{list}^t is weakly connected and stays weakly connected during the computation after some state s_{t_0} at every time $t \geq t_0$. By Lemma 3.3.2 and 3.3.3 follows that all edges in G_{list}^t are shortened over time. Note that as soon as a node receives a forward message there has to be a state s_{t_1}, $t_1 \geq t_0$, with at least one edge in E_{e-list}^{t1}. According to Lemma 3.3.4 eventually no edge in $E_{e-list}^{t_1}$ can be shortened and either (u, v) and $(v, u) \in E_{e-list}^{t_2}$ at some state s_{t_2} at time $t_2 \geq t_1$ or a new edge is added to $E_{e-list}^{t_1}$. As for each node there are at most two edges in E_{e-list}, eventually all edges have to be added and a state s_{t_3} at time $t_3 \geq t_2$ is reached with $(u, v) \in E_{e-list}^{t_3} \Rightarrow (v, u) \in E_{e-list}^{t_3}$. According to Lemma 3.3.5 there is a state s_{t_4} such that in every state $s_{t'}$ after $(u, v) \in E_{e-list}^{t'} \Rightarrow (v, u) \in E_{e-list}^{t'}$. Now, according to Lemma 3.3.6 the computation contains a later state s_{t_5} at time $t_5 \geq t_4$ such that $E_{e-list}^{t_5}$ is strongly connected. Then by applying lemma 3.3.7 $G_l^{t_5} ist$ is a solution for the sorted-list problem.

Our second theorem claims that after a sorted list is formed the computation reaches a state, such that a cyclic list is a sub-graph of the graph of explicit edges.

Theorem 3.3.4 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLD}$ then eventually the computation reaches a state s_t such that $G_{e-cycle}^t$ is a cyclic list.*

Proof. According to Theorem 3.3.3 the computation reaches a state s_{t_0} at a time t_0 such that $G_{e-list}^{t_0}$ is a sorted list. Then only v_{min} and v_{max} with $h(v_{min}) = \min\{h(v) : v \in V\}$ and $h(v_{max}) = \max\{h(v) : v \in V\}$ are missing a predecessor or successor. According to the protocol $P^{SMALL-WORLD}$ $v_{min}.cycle \neq nil$ and $v_{max}.cycle \neq nil$. Let $w = v_{min}.cycle$ then $h(w) > h(v_{min})$ and w receives eventually a cycle message and responds with a response-cycle message containing w' with $h(w') > h(w)$ and $w' = w.successor$ or $w' = w.long-range$. Then in some state s_{t_1} at time $t_1 \geq t_0$ v_{min} will receive the message $m = (response - cycle, w')$ and thus in s_{t_1} $v_{min}.cycle \geq w'$. Obviously $v_{min}.cycle$ can only be updated a finite number of times and eventually a state s_{t_2} is reached such that no update takes place in a later state. Then at time $t_2 \geq t_1$ and in every state afterwards $v_{min}.cycle = v_{max}$. By symmetric arguments we can show that there is also a state s_{t_3} at time $t_3 \geq t_2$ such that $v_{max}.cycle = v_{min}$ and $v_{min}.cycle = v_{max}$ in every state s_t, $t \geq t_3$. □

Theorem 3.3.5 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLD}$ then eventually the computation reaches a state s_t such that G_e^t is a small-world network.*

Proof. From Theorems 3.3.3 and 3.3.4 follows that the computation of $P^{SMALL-WORLD}$ reaches a state s_{t_0} such that in every state s_t, $t \geq t_0$, G_e^t contains the 1- dimensional lattice (the cyclic list). Then there is a later state s_{t_1} at time $t_1 \geq t_0$ in the computation such that all long-range links have been forgotten at least once between these states. It can easily been shown by using properties shown in [14] that the maximal age of a long-range link is $\mathcal{O}(n)$ w.h.p.(with high probability is a probability $\geq 1 - \frac{1}{n^c}$). In each update a long-range link is forgotten with a probability at least $\frac{1}{u.age}$ (for $u.age \geq 3$). Then $\prod_{u.age=3}^{n}(1 - \frac{1}{u.age}) = \frac{2}{n}$ is the probability that a long-range link is not forgotten n times. And thus w.h.p. $u.age = \mathcal{O}(n)$.

Then it can also been proven that there is a state s_{t_1} such that each node updated its long-range link at least $\mathcal{O}(n)$ times and thus w.h.p. all long-range links have been forgotten at least once. After this state the move-and forget process for the long range links is performed on the cyclic list and eventually a state $s_{t'}$ at time $t' \geq t_1$ is reached such that in any later state s_t, $t \geq t'$, G_e^t forms a small-world network according to [14]. □

Then Theorem 3.3.2 follows immediately.

Closure

We will now show that for $P^{SMALL-WORLD}$ also the closure property holds.

Theorem 3.3.6 *If in an initial graph $G \in SMALL - WORLD$ every node executes the protocol $P^{SMALL-WORLD}$ then each possible computation leads to a graph $G' \in SMALL - WORLD$.*

Proof. If $G_e^t \in SMALL - WORLD$ then explicit edges in $G_{e-cycle}^t$ do not change, as edges $(u, u.successor)$ (resp. $(u, u.predecessor)$) only change if u receives a node w with $h(u) < h(w) < h(u.successor)$ (resp. $h(u) > h(w) > h(u.predecessor)$). As the edges $(u, u.successor)$ and $(u, u.predecessor)$ form a sorted list such a node w can not exist. Edges $(u, u.cycle)$ with $h(u) < h(u.cycle)$ (resp. $h(u) > h(u.cycle)$) only change if u receives a node w with $h(w) > h(u.cycle)$ or $h(w) < h(u)$ (resp. $h(w) < h(u.cycle)$ or $h(w) > h(u)$). As only the nodes with minimal and maximal position v_{min} and v_{max} have a cycle edge with $v_{min}.cycle = v_{max}$ and $v_{max}.cycle = v_{min}$ such a node w can not exist. The only explicit edges that change are long-range links. As according to [14] the distribution of the lengths of long-range links is stationary by the the Move & Forget process the small-world property is maintained. □

3.3.7 Protocol P$^{\text{SMALL-WORLDsync}}$

In the next section we consider the efficiency of $P^{SMALL-WORLD}$ and therefore analyze it according to our STSS model. In fact we will adapt the protocol to the synchronous setting. As we have seen for the sorted list slight modifications can lead to improved bounds on the stabilization and maintenance work. We do not describe the protocol $P^{SMALL-WORLDsync}$ in detail as the basic ideas are the same as in $P^{SMALL-WORLD}$. The only changes are that we do not react on each message individually, but can react on a batch of messages received in the same round. In this way we can ensure that we update a long-range link only once in one round and that all nodes that should be forwarded are sorted according to their position and thus a node only sends one forward message to another node. We therefore collect all received identifiers in move-forget messages in one set LRL, all identifiers received by forward or introduction messages in a set S and all identifiers received by cycle messages in a set $Cycle$. We

further use the modification we introduced in the improved $P^{LISTsync}$ to prevent nodes from introducing themselves to nodes that can't be their successors or predecessors. For simplicity we assume that $u.\tau$ is true in every round.

We present a pseudo code implementation of $P^{SMALL-WORLDsync}$ including a probing mechanism in Algorithm 3.3.11-3.3.19.

Algorithm 3.3.11 $P^{SMALL-WORLDsync}$

```
message m ∈ u.Ch →
S = ∅                                  ▷ Ids of nodes that should be forwarded by forward messages
LRL = ∅                                ▷ Ids of candidates for the long-range link
Cycle = ∅          ▷ Set of nodes that should be responded to by respond-cycle-virtual messages
Probing = ∅                            ▷ Set of nodes that should be forwarded by probing messages
while m ∈ u.Ch do
    if type=forward then
        S.insert(v)
    else if type=introduction then
        S.insert(v)
    else if type=long-range then
        long-range(v)
    else if type=probing-right then
        Probing.insert(v)
    else if type=probing-left then
        Probing.insert(v)
    else if type=cycle then
        Cycle.insert(v)
    else if type=response-cycle then
        response-cycle(v)
    else if type=move-forget then
        if v2 = u.long − range then
            LRL.insert(v0)
            LRL.insert(v1)
        else
            S.insert(v0)
            S.insert(v1)
            S.insert(v2)
send-cycle()
send-probes()
move-forget()
send-list()
τ →
introduction()
probing()
```

Algorithm 3.3.12 LONG-RANGE(V)

if $u.predecessor \neq nil \wedge u.successor \neq nil$ **then**
  send message $m = (move - forget, u.predecessor, u.successor, u)$ to v
else if $u.predecessor \neq nil$ **then**
  send message $m = (move - forget, u.predecessor, u.cycle, u)$ to v
else if $u.successor \neq nil$ **then**
  send message $m = (move - forget, u.cycle, u.successor, u)$ to v
else
  S.insert(v)

Algorithm 3.3.13 RESPONSE-CYCLE(v)

if $u.predecessor = nil \wedge (h(v) > h(u.cycle) > h(u) \vee (u.cycle = nil \wedge h(v) > h(u)))$ **then**
  if $u.cycle \neq nil$ **then**
    S.insert($u.cycle$)
  $u.cycle = v$
else if $u.successor = nil \wedge (h(v) < h(u.cycle) < h(u) \vee (u.cycle = nil \wedge h(v) < h(u)))$ **then**
  if $u.cycle \neq nil$ **then**
    S.insert($u.cycle$)
  $u.cycle = v$
else
  S.insert(v)

3.3.8 Analysis in the STSS model

Analyzing the same protocol $P^{SMALL-WORLDsync}$ in the STSS model we show some bounds on the complexities defined in Chapter 2.

Theorem 3.3.7 *If an initial graph $G^{IT} \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLDsync}$ then the graph converges to a graph $G \in SMALL - WORLD$ (Convergence) with a stabilization time of $\mathcal{O}(n + \delta)$ and a stabilization work of $\mathcal{O}(n^2)$. If in an initial graph $G \in SMALL - WORLD*$ every node executes the protocol $P^{SMALL-WORLDsync}$ then each possible computation leads to a graph $G' \in SMALL - WORLD$ (Closure) with a maintenance work of $\mathcal{O}(n)$.*

Stabilization Time

We now prove an upper bound on the stabilization time of $P^{SMALL-WORLDsync}$. The stabilization time is bounded by the number of nodes n and the *convergence time* of the move-forget process δ

Theorem 3.3.8 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLDsync}$ then the graph converges to a graph $G' \in SMALL - WORLD$ (Convergence) with a stabilization time of $\mathcal{O}(n + \delta)$ w.h.p..*

Like the convergence in the ATSS model we will prove the stabilization time in the STSS model by proving the stabilization time for some intermediate steps. We begin with the following observation: After at most two rounds each node has received every initial message and every response to an initial message.

Algorithm 3.3.14 SEND-CYCLE()

if $Cycle \neq \emptyset$ **then**
  $W = \{u.successor, u.predecessor, u.cycle, u.long - range\}$
  Sort all nodes v in $Cycle$ according to their position $h(v)$, such that $h(v_{-k}) < h(v_{-(k-1)}) < \cdots < h(v_{-1}) < h(u_i) < h(v_1) < \cdots < h(v_{k'-1}) < h(v'_k)$
  for $j = 1$ to $k' - 1$ **do**
    Send message $m' = (forward, v_{j+1})$ to v_j
  if $\exists w : h(w) > h(v'_k) \wedge w \in W$ **then**
    $w =$ argmin $\{h(w') : h(w') > h(v'_k) \wedge w' \in W\}$
    send message $m = (forward, w)$ to v'_k
  else if $\exists w : h(w) < h(v) \wedge w \in W$ **then**
    $w =$ argmin $\{h(w') : h(w') < h(v) \wedge w' \in W\}$
    send message $m = (response - cycle, w)$ to v
  else
    send message $m = (response - cycle, u)$ to v
  for $j = 1$ to $k - 1$ **do**
    Send message $m' = (forward, v_{-(j+1)})$ to v_{-j}
  if $\exists w : h(w) < h(v_k) \wedge w \in W$ **then**
    $w =$ argmax $\{h(w') : h(w') < h(v_k) \wedge w' \in W\}$
    send message $m = (forward, w)$ to v'_k
  else if $\exists w : h(w) > h(v) \wedge w \in W$ **then**
    $w =$ argmax $\{h(w') : h(w') > h(v) \wedge w' \in W\}$
    send message $m = (response - cycle, w)$ to v
  else
    send message $m = (response - cycle, u)$ to v

Algorithm 3.3.15 SEND-PROBES()

if $Probing \neq \emptyset$ **then**
  Sort all ids v in $Probing \cup \{u.successor, u.predecessor, u.long - range\}$ according to their position $h(v)$, such that $h(v_{-k}) < h(v_{-(k-1)}) < \cdots < h(v_{-1}) < h(u) < h(v_1) < \cdots < h(v_{k'-1}) < h(v'_k)$
  for $j = 1$ to $k' - 1$ **do**
    if $v_{j+1} \notin \{u.successor, u.predecessor, u.long - range\}$ **then**
      Send message $m' = (probing, v_{j+1})$ to v_j
  S.insert(v_1)
  for $j = 1$ to $k - 1$ **do**
    if $v_{-(j+1)} \notin \{u.successor, u.predecessor, u.long - range\}$ **then**
      Send message $m' = (probing, v_{-(j+1)})$ to v_{-j}
  S.insert(v_{-1})

Algorithm 3.3.16 MOVE-FORGET()

if $M \neq \emptyset$ **then**
  Choose one element v_i in LRL uniformly at random
  S.insert($LRL \cup \{u.long - range\} - \{v_i\}$)
  $u.long - range = v_i$
  $reset = true$ with probability $\phi(u.age)$
  if reset=true **then**
    S.insert($u.long - range$)
    $u.long - range = u$

Algorithm 3.3.17 SEND-LIST()

if $S \neq \emptyset$ **then**
  Sort all ids v in $S \cup u.successor \cup u.predecessor \cup u.long - range$ according to their position $h(v)$, such that $h(v_{-l}) < h(v_{-(l-1)}) < \cdots < h(v_{-1}) < h(u) < h(v_1) < \cdots < h(v_{k-1}) < h(v_k)$
  $u.successor = v_1$
  for i=1 to k-1 **do**
    if $i \neq 1$ or $v_2 \neq u.long - range$ **then**
      Send message $m' = (forward, v_{i+1})$ to v_i
      if v_i is received by a message $m = (introduction, v_i)$ **then**
        send message $m' = (forward, v_{i-1})$ to v_i
  $u.predecessor = v_{-1}$
  for i=1 to l-1 **do**
    if $i \neq 1$ or $v_{-2} \neq u.long - range$ **then**
      Send message $m' = (forward, v_{-(i+1)})$ to v_{-i}
      if v_{-i} is received by a message $m = (introduction, v_{-i})$ **then**
        send message $m' = (forward, v_{-(i-1)})$ to v_{-i}

Algorithm 3.3.18 INTRODUCTION()

if $h(u.predecessor) \neq nil$ **then**
  send message $m = (introduction, u)$ to $u.predecessor$
else
  send message $m = (cycle, u)$ to $u.cycle$
if $h(u.successor) \neq nil$ **then**
  send message $m = (introduction, u)$ to $u.successor$
else
  send message $m = (cycle, u)$ to $u.cycle$
send message $m = (long - range, u)$ to $u.long - range$

Algorithm 3.3.19 PROBING()

if $h(u.long - range) < h(u)$ **then**
  if $h(u.long - range) \leq h(u.predecessor)$ **then**
    send message $m = (probing - left, u.long - range)$ to $u.predecessor$
else
  if $u.long - range \geq u.successor$ **then**
    send message $m = (probing - right, u.long - range)$ to $u.successor$
if $u.cycle \neq nil \wedge h(u.cycle) < h(u.predecessor)$ **then**
  send message $m = (probing - left, u.cycle)$ to $u.predecessor$
else if $u.cycle \neq nil \wedge h(u.cycle) > h(u.successor)$ **then**
  send message $m = (probing - right, u.cycle)$ to $u.successor$

Then each node u only receives long-range-respond messages from nodes w with $w = u.long - range$ and cycle-respond messages from nodes w with $w = u.cycle$. We call this point of time by t_0 with $t_0 \leq 2$. Then at every time $t \geq t_0$ for each $w = u.long - range$ or $w = u.cycle$ u has started a probing.

Theorem 3.3.9 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLDsync}$ then the graph converges to a graph G' such that G'_{e-list} is a sorted list after $\mathcal{O}(n)$ rounds.*

To show this theorem we need to introduce a different notion of connectivity.

Definition 3.3.8 *We call two nodes u and v* connected over time *at time t if there is a* over time connecting path $p = (u = w_0, w_1, w_2, \cdots, w_l = v)$ *and $(w_i, w_{i+1}) \in E_{list}$ at time $t + i$ $\forall i < l$. That means that u and v don't have to be connected all the time but it is possible to send a message from u to v at time t along edges in E_{list}. We further call two nodes weakly connected over time at time t, if there a weakly over time connecting path $p = (u = w_0, w_1, w_2 \cdots, w_l = v)$ and w_i and w_{i+1} or w_{i+1} and w_i are connected over time.*

In order to show Theorem 3.3.9 we first show a helpful lemma.

Lemma 3.3.8 *If $(u, v) \in E^t$ or u and v are connected over time at time t then u and v are connected over time for every time $t' > t$.*

Proof. In the following we assume $h(u) < h(v)$, as for the case $h(v) < h(u)$ symmetric arguments can be applied. We first consider the case that $(u, v) \in E^t$. Then either v is stored in a message in $u.Ch$ or v is stored in an internal variable. If v is stored in an internal variable then $v = u.successor$ or $v = u.long - range$ or $v = u.cycle$. We first show that u and v are connected over time if u keeps v in its internal variables and then proceed with the case that v is delegated. If $v = u.successor$ and v is not delegated then obviously u and v are connected over time and stay connected over time. If $v = u.long - range$ or $v = u.cycle$ and v is not delegated then according to our observation u has started a probing to v. Then this probing message takes a certain path through the network and u and v are connected over time along this probing path. A probing message received by a node w is only forwarded to $w.successor$, $w.long - range$ or a node w' with $h(u) < h(w) < h(w') < h(v)$ that w also received a probing message for.

If the probing message is forwarded to $w.successor$ then w and $w.successor$ are connected and thus connect u and v over time. If the probing message is forwarded to $w.long - range$ then according to our observation w also sends a probing message destined for $w.long - range$ and w and $w.long - range$ and thus also u and v are connected over time along the corresponding probing path. The same arguments hold for the case that the probing message is forwarded to another probed node w'. By induction follows that u and v are connected over time at time t and at every time after as long as u stores v in its internal variables.

If v is not longer stored in an internal variable but is delegated, then v is delegated to a node w that is connected to u by edges in E_{list} in the send-list action, or v is delegated to $u.long - range$ for which we already know that u and $u.long - range$ are connected over time. If v is stored in a message in $u.Ch$, then when u receives the message either v is stored in an internal variable or is delegated. For both cases we have already shown that u and v then are connected over time.

Then also follows that if u and v are connected over time at time t they stay connected over time at every time $t' > t$. □

We are now ready to prove Theorem 3.3.9.

Proof. As the network is initially weakly connected and stays weakly connected according to Lemma 3.3.1 also G_{list} is weakly connected over time and stays weakly connected over time one round after $t_0 \leq 2$ according to Lemma 3.3.8. In the following we reuse the analysis presented in the proof of Theorem 3.2.5, where we showed the stabilization time for P^{LIST}.

Let v_i and v_{i+1} be two arbitrary consecutive nodes in the sorted list. Then there is a weakly over time connecting path p for v_i and v_{i+1}. We show that the minimal range of all weakly over time connecting paths is decreasing, such that the two nodes are directly connected after at most $\mathcal{O}(n)$ synchronous rounds.

Let p^t be a weakly over time connecting path for two arbitrary consecutive nodes v_i and v_{i+1} at time $t \geq t_0$. We then show that we can find a weakly over time connecting path at time $t' > t$ with a strictly smaller range. We firstly prove that if one of the border nodes w^t_{min} and w^t_{max} defining the range of p_t has two outgoing edges in $E^t \cap p^t$, then we can construct a connecting path p^{t+1} with a smaller range. We will show this for w^t_{min} as the same arguments can be applied for w^t_{max}.

If w^t_{min} has two outgoing edges $(w^t_{min}, x), (w^t_{min}, y) \in p^t$ then we can construct a new weakly over time connecting path p^{t+1}. From the proof of Lemma 3.3.8 follows that there is an over time connecting path from w^t_{min} to x and y over $w^t_{min}.successor$. Thus in p^{t+1} w^t_{min} can be substituted by $w^t_{min}.successor$ and the range of the weakly over time connecting path decreases. It remains to show that it does not take too long before a border node has two outgoing edges. In case w^t_{min} does not have two outgoing edges we construct p^{t+1} out of p^t in the following way.

If $(x, y) \in p^t$ with $h(x) < h(y)$ and y is delegated then according to Lemma 3.3.8 there is an over time connecting path $p' = (x = w_0, w_1 = x.successor, w_2, \cdots w_l = y)$ with $h(x) \leq h(w_i) \leq h(y)$ from x to y. Thus we can substitute (x, y) by p' in p^{t+1}. If $(x, y) \in p^t$ with $h(x) < h(y)$ and y is not delegated then we simply keep $(x, y) \in p^{t+1}$.

If $(x, y) \in p^t$ with $h(x) > h(y)$ and y is delegated then according to Lemma 3.3.8 there is an over time connecting path $p' = (x = w_0, w_1 = x.predecessor, w_2, \cdots w_l = y)$ with $h(x) \geq h(w'_i) \geq h(y)$ from x to y. Thus we substitute (x, y) by p' in p^{t+1}. If $(x, y) \in p^t$ with $h(x) > h(y)$ and y is not delegated, then $y = x.predecessor$ and x introduces itself to y and we substitute $(x, y) \in p^t$ by $(y, x) \in p^{t+1}$.

By using this construction scheme we can show that it takes at most $\max\{0, n - (t - t_0)\}$ rounds before w^t_{min} has two outgoing edges on a path p^t. We prove this by induction on t. Let p^0 be an arbitrary

weakly over time connecting path for two consecutive nodes x_i, x_{i+1} in the sorted list. Then $w_{min}^{t_0}$ is the left border node of this path. If $w_{min}^{t_0}$ already has two outgoing edges on p^{t_0} then it takes obviously most $\max\{0, n\}$ rounds. Otherwise there is at least one incoming edge $(y, w_{min}^{t_0})$ on p^{t_0}. For each such incoming edge $(y, w_{min}^{t_0})$ the node y either introduces itself to $w_{min}^{t_0}$ or delegates $w_{min}^{t_0}$. In the first case $w_{min}^{t_0}$ then has an outgoing edge in $p^{t_1=t_0+1}$ and every following path $p^{t'}$, $t' \geq t_1$, according to our construction scheme. In the second case (y, w_{min}^{t-0}) is substituted by a path p' as described above. Then $w_{min}^{t_0}$ still has an incoming edge $(y', w_{min}^{t_0})$, but with $h(y') < h(y)$. Obviously $w_{min}^{t_0}$ can only be delegated n times before $(y', w_{min}^{t_0})$ is substituted by an outgoing edge.

Let p^t be an undirected path connecting v_i, v_{i+1} in the sorted list that was constructed out of p^{t_0} according to our construction scheme. If w_{min}^t already has two outgoing edges on p^t then it takes obviously most $\max\{0, n-(t-t_0)\}$ rounds. Otherwise there is at least one incoming edge (y, w_{min}^t) on p^t. Then $(y, w_{min}^t) \notin p^{t-1}$ but w_{min}^t was delegated to y by another node z with $h(z) > h(y)$, otherwise y would have introduced itself to w_{min}^t and w_{min}^t would have an outgoing edge instead. Thus $(z, w_{min}^t) \in p^{t-1}$. For this edge the same observation holds. As w_{min}^t has been delegated in every round before, w_{min}^t has been delegated t times by nodes z with $h(z) > h(y)$. Thus there are at most $n-(t-t_0)$ nodes w' with $h(w_{min}^t) < h(w') < h(y)$ that w_{min}^t can still be delegated to before the edge is substituted by an outgoing edge. Then for a path p^t it takes at most $\max\{0, n-(t-t_0)\}$ rounds before w_{min}^t has two outgoing edges and if w_{min}^t has two outgoing edges we can construct a path with a smaller range. Thus after $\mathcal{O}(n)$ rounds v_i, v_{i+1} are directly connected, i.e. $(v_i, v_{i+1}) \in E_{e-list}$ and $(v_{i+1}, v_i) \in E_{e-list}$. □

Theorem 3.3.10 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLDsync}$ then the graph converges to a graph G' such that $G'_{e-cycle}$ is a cyclic list after $\mathcal{O}(n)$ rounds.*

Proof. The proof is based on the same arguments we already used in the analysis of the convergence in the ATSS model for Theorem 3.3.4.

According to Theorem 3.3.9 the computation reaches a state s_{t_0} at a time t_0 such that $G_{e-list}^{t_0}$ is a sorted list after $\mathcal{O}(n)$ rounds. Then only v_{min} and v_{max} with $h(v_{min}) = \min\{h(v) : v \in V\}$ and $h(v_{max}) = \max\{h(v) : v \in V\}$ are missing a predecessor or successor. Thus according to the protocol $P^{SMALL-WORLDsync}$ $v_{min}.cycle \neq nil$ and $v_{max}.cycle \neq nil$. Let $w = v_{min}.cycle$ then $h(w) > h(v_{min})$ and w receives a cycle message in the next round and responds with a response-cycle message containing w' with $h(w') > h(w)$ and $w' = w.successor$ or $w' = w.long - range$. Then v_{min} will receive the message $m = (response - cycle, w')$ and thus after at most two rounds $v_{min}.cycle \geq w'$. Obviously $v_{min}.cycle$ is updated in every round as long as $w' < v_{max}$. Then after $\mathcal{O}(n)$ rounds $v_{min}.cycle = v_{max}$. By symmetric arguments we can show that also after $\mathcal{O}(n)$ rounds $v_{max}.cycle = v_{min}$ and $v_{min}.cycle = v_{max}$ in every state afterwards. □

Theorem 3.3.11 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{SMALL-WORLDsync}$ then the graph converges to a graph $G' \in SMALL-WORLD$ after $\mathcal{O}(n+\delta)$ rounds w.h.p..*

Proof. The proof is based on the same arguments we already used in the analysis of the convergence in the ATSS model for Theorem 3.3.5.

From Theorem 3.3.10 follows that computation of $P^{SMALL-WORLDsync}$ reaches a state s_{t_0} such that in every state s_t, $t \geq t_0$, $G_{e-cycle}^t$ contains the cyclic list after $\mathcal{O}(n)$ rounds. It can easily been shown

by using properties shown in [14] that the maximal age of a long-range link is $\mathcal{O}(n)$ w.h.p.(with high probability is a probability $\geq 1 - \frac{1}{n^c}$). Thus w.h.p. a long-range link is forgotten after $\mathcal{O}(n)$ rounds. As soon as each long-range link was forgotten after t_0 the move-and forget process for the long range links is performed on the cyclic list and after $\mathcal{O}(\delta)$ rounds G_e^t forms a small-world network and maintains it forever according to [14]. □

Then Theorem 3.3.8 follows immediately. We conclude the analysis by showing the Closure property in the STSS model.

Theorem 3.3.12 *If in an initial graph $G \in SMALL - WORLD$ every node executes the protocol $P^{SMALL-WORLDsync}$ then each possible computation leads to a graph $G' \in SMALL - WORLD$ (Closure).*

Proof. The proof is the same as in the asynchronous setting for Theorem 3.3.6. □

Stabilization Work

Theorem 3.3.13 *If in an initial graph $G \in IT$ every node executes $P^{SMALL-WORLDsync}$ then each possible computation leads to a graph $G' \in SMALL - WORLD$ with a stabilization work of $\mathcal{O}(n^2)$.*

Proof. In each round a node sends at most $\mathcal{O}(1)$ forward-messages to another node in the send-list action. Then each node also receives at most $\mathcal{O}(1)$ forward-messages by on other node in one round. Thus each node receives at most $\mathcal{O}(n)$ forward-messages in one round. Then according to Theorem 3.3.8 each node receives and sends at most $\mathcal{O}(n^2 + n\delta)$ forward messages during the stabilization.

Each node u sends at most $\mathcal{O}(1)$ introduction message in one round to $u.successor$ and $u.predecessor$. Each node receives at most one introduction messages from a node w with $w \neq u.successor$ and $w \neq u.predecessor$. Then each node sends and receives at most $\mathcal{O}(n + \delta)$ introduction messages.

Each node sends at most one long-range message in one round. As the long-range links perform a random walk on the cyclic list, there always exists the possibility for one node u that there are $\mathcal{O}(n)$ nodes v with $u = v.long - range$. Then a node u can receive up to $\mathcal{O}(n)$ long-range messages in one round. Then the number of received and sent long-range messages can only be bounded by $\mathcal{O}(n^2 + n\delta)$.

Each node initiates at most $\mathcal{O}(1)$ probing messages in each node. Each node sends at most $\mathcal{O}(1)$ probing messages to another node in one round. Then each node also receives at most $\mathcal{O}(1)$ probing message from another node in one round. Then each node sends or receives at most $\mathcal{O}(n^2 + n\delta)$ probing messages during the stabilization.

Each node sends at most $\mathcal{O}(1)$ cycle message in one round and receives a cycle message once from a node v which is not the node with the minimal or maximal position sending a cycle message to u. If u receives cycle messages from two nodes v and w and $h(v) < h(w) < h(u)$ (resp. $h(v) > h(w) > h(u)$) then u will send a message $m = (forward, v)$ to w thus w updates its predecessor (resp. successor) and will not send an cycle message to u again. Then each node u sends and receives at most $\mathcal{O}(n + \delta)$ cycle messages. Furthermore u also only sends and receives $\mathcal{O}(n + \delta)$ response-cycle messages. □

Maintenance Work

Theorem 3.3.14 *If in an initial graph $G \in SMALL - WORLD*$ every node executes the protocol $P^{SMALL-WORLDsync}$ then each possible computation leads to a graph $G' \in SMALL - WORLD$ with a maintenance work of $\mathcal{O}(n)$.*

Proof. As we already showed in the proof of Theorem 3.3.13 each node sends and receives at most $\mathcal{O}(n)$ messages of each message type. We can further show by a more detailed analysis that each node only sends and receives a polylogarithmic number of probing messages in expectation. In analysis in [14] the authors claim that at each routing step, the distance to the destination is halved with a probability of at least $\Omega\left(\frac{1}{ln^{1+\epsilon}d}\right)$ with d being the distance to the destination. The proof is based on the following idea: B is the set of nodes v, which have a distance to the destination node z which is smaller than half of the distance of the current node w (of the routing path) to the destination, i.e. $B = \{v \in \mathbb{Z}^k : dist(v,z) \geq dist(w,z)/2\}$. Then the authors show that in the routing such a node v is reached by w with the aforementioned probability $\Omega\left(\frac{1}{ln^{1+\epsilon}d}\right)$. So, in our 1-dimensional case, B would consist of all the nodes being at most $d/2$ away from the destination, both on the left and on the right of the destination node, where d is the distance of the current node to the destination. Since in our probing procedure from u to $u.long - range$, we allow the usage of long-range links only if they are at the left side of $u.long - range$, the probability that at any routing step the distance to the destination would be halved is now equal to the probability that the next node of the probing path lies in $A = \{v \in \mathbb{Z}^k : dist(v,z) \geq dist(w,z)/2 \wedge h(v) < h(z)\}$. We restrict the set B by the additional constraint $h(v) < h(z)$. Then $|A| \geq \frac{|B|}{2}$. Then also the probability to reach a node in A instead of B is halved and thus still $\Omega\left(\frac{1}{ln^{1+\epsilon}d}\right)$. So, by the analysis of Theorem 2 in [14], we get that the expected number of steps for a probing message to reach its destination in a stable state is $O(ln^{2+e}d)$. □

Single Join and Leave Events

For the completeness of the analysis we consider dynamic changes of the network in terms of single joining and leaving nodes.

Theorem 3.3.15 *If a node joins or leaves a network that is previously in $SMALL - WORLD$ it takes at most $\mathcal{O}(ln^{2+\epsilon}n)$ rounds until a legal state is reached again with an additional work of $\mathcal{O}(1)$ for each node per round.*

Proof. If a node u joins the network and is connected to an arbitrary node v already in the network, u introduces itself to v. Then v either stores u in its internal variable if $u = v.successor$ or $u = v.predecessor$ or v forwards u according to the greedy routing scheme we already described in the analysis of the probing procedure in the proof of Theorem 3.3.14. Then it takes at most $\mathcal{O}(ln^{2+\epsilon}n)$ rounds until u is connected to its correct predecessor and successor. Thus the cyclic list is stabilized.

When a node u leaves the network the connections it had to and from other nodes are removed. As a consequence, two nodes v and w formerly $u.predecessor$ and $u.successor$ have no right and left neighbors. These nodes then initiate cycle messages. Also these cycle messages are forwarded according to the greedy routing scheme and thus after at most $\mathcal{O}(ln^{2+\epsilon}n)$ until v and w will be directly connected.

As we only add or remove one node the distribution of long-range links is still stationary and resembles a harmonic distribution if n is large enough. □

3.4 Resource Discovery or a self-stabilizing Clique

3.4.1 Introduction

To perform cooperative tasks in distributed systems the network nodes have to know which other nodes are participating. Examples for such cooperative tasks range from fundamental problems such as group-

based cryptography [58], verifiable secret sharing, distributed consensus [64], and broadcasting [67] to peer-to-peer (P2P) applications like distributed storage, multiplayer online gaming, and various social network applications such as chat groups. To perform these tasks efficiently knowledge of the complete network for each node is assumed. Considering large-scale, real-world networks this complete knowledge has to be maintained despite high dynamics, such as joining or leaving nodes, that lead to changing topologies. Therefore the nodes in a network need to learn about all other nodes currently in the network. This problem called *resource discovery*, i.e. the discovery of the addresses of all nodes in the network by every single node, is a well studied problem and was firstly introduced by Harchol-Balter, Leighton and Lewin in [25].

As mentioned in [25] the resource discovery problem can be solved by a simple swamping algorithm also known as *pointer doubling*: in each round, every node informs all of its neighbors about its entire neighborhood. While this just needs $O(\log n)$ communication rounds to inform every node about any other node in every weakly connected network of size n, the work spent by the nodes can be very high and far from optimal. Moreover, in the stable state (i.e., each node has complete knowledge) the work spent by every node in a single round is $\Theta(n^2)$, which is certainly not useful for large-scale systems.

Alternatively, each node may just introduce a single neighbor to all of its neighbors in a round-robin fashion. However, it is easy to construct initial situations in which this strategy is not better than pointer doubling in order to reach complete knowledge. The problem in both approaches is the high amount of redundancy: addresses of nodes may be sent to other nodes that are already aware of that address.

In [25] a randomized algorithm called the *Name-Dropper* is presented that solves the resource discovery problem within $\mathcal{O}(\log^2 n)$ rounds w.h.p. and work of $\mathcal{O}(n^2 \log^2 n)$. In [45] a deterministic solution for resource discovery in distributed networks was proposed by Kutten et al. Their solution uses the same model as in [25] and improves the number of communication rounds to which takes $\mathcal{O}(\log n)$ rounds and $\mathcal{O}(n^2 \log n)$ amount of work.

Konwar et al. presented solutions for the resource discovery problem considering different models, i.e. multicast or unicast abilities and messages of different sizes, where the upper bound for the work is $O(n^2 \log^2 n)$. In their algorithms they also considered when to terminate, i.e. how can a node detect that its knowledge is already complete.

Recently resource discovery has been studied by Haeupler et. al. in [24], in which they present two simple randomized algorithms based on gossiping that need $\Omega(n \log n)$ time and $\Omega(n^2 \log n)$ work per node on expectation. They only allow nodes to send a single message containing at most one address of size $\log n$ in each round. Thus their model is more restrictive compared to the model used in [25, 45] and leads to an increased runtime in the number of rounds.

We present a deterministic solution that follows the idea of [24] and limits the number of messages each node has to send and the number of addresses transmitted in one message. Our goal is to reduce the number of messages sent and received by each node such that we avoid nodes to be overloaded. In detail we show that resource discovery can be solved in $\mathcal{O}(n)$ rounds and it suffices that each node sends and receives $\mathcal{O}(n)$ messages in total, each message containing $\mathcal{O}(1)$ addresses. Our solution is the first solution for resource discovery that not only considers the total number of messages but also the number of messages a single node has to send or receive.

Note that $\Omega(n)$ is a trivial lower bound for the work of each node to gain complete knowledge: starting with a list, in which each node is only connected to two other nodes, each node has to receive at least $n - 3$ IDs. So our algorithm is worst case optimal in terms of message complexity. Furthermore our algorithm can handle the deletion of edges and joining or leaving nodes, as long as the graph remains weakly connected. Modeling the current knowledge of all nodes as a directed graph, i.e. there is an

edge (u, v) iff u knows v's identifier, one can think of resource discovery as building and maintaining a complete graph, a clique, as a virtual overlay network. We therefore present a protocol P^{CLIQUE} that topological self-stabilizing solves the overlay problem $CLIQUE$, in which the nodes form a complete graph starting from an initially weakly connected graph. An example of a clique is given in Figure 3.14.

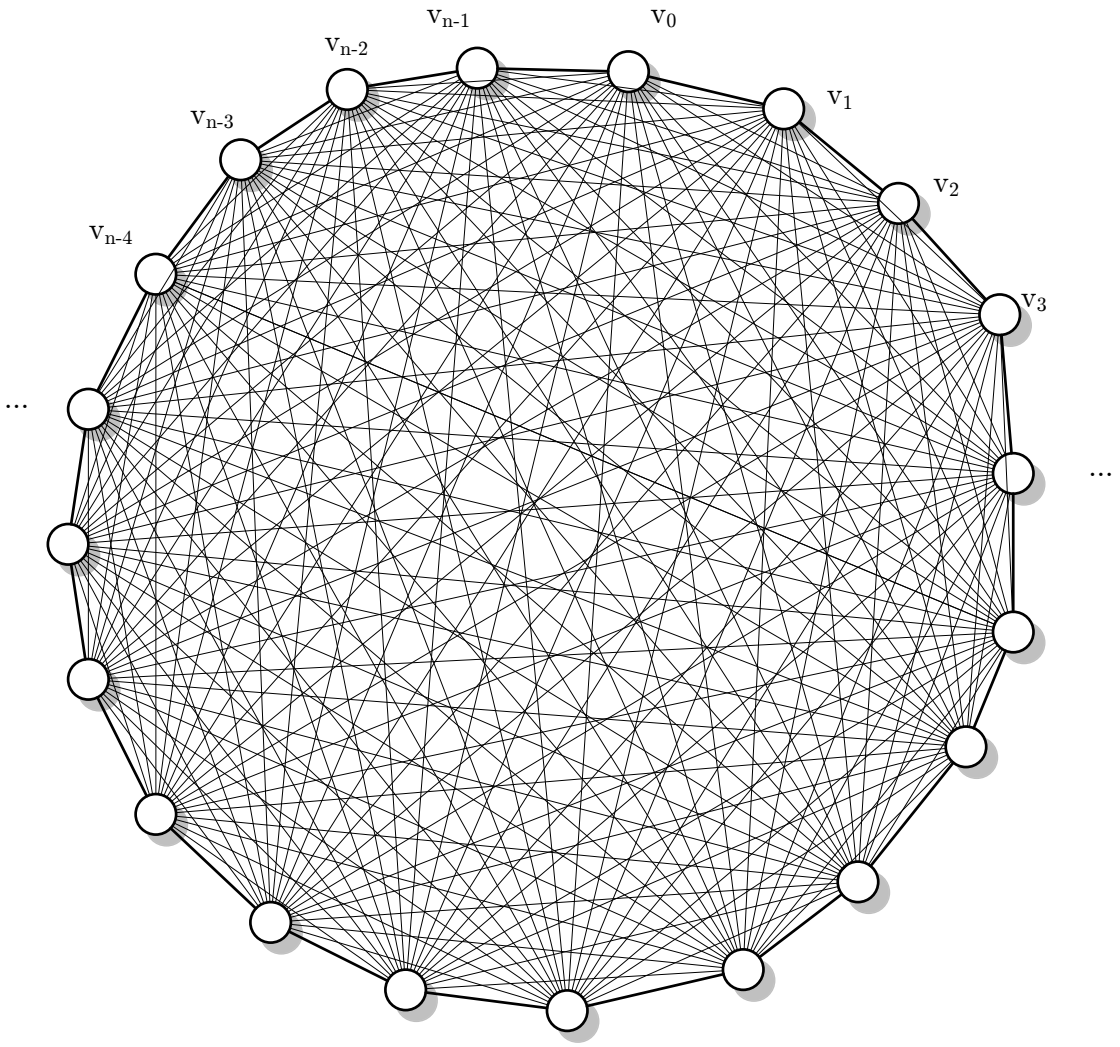


Figure 3.14: Example for a clique network

A self-stabilizing spanning tree algorithm could be used as an intermediate step to build a complete graph. A large number of self-stabilizing distributed algorithms has already been proposed for the formation of spanning trees in static network topologies, [10, 9, 27]. For example in [10] the authors present a self-stabilizing spanning tree with minimal degree for the given network and in [9] a fast algorithm for a self-stabilizing spanning tree is presented, which reaches optimal convergence time of $\mathcal{O}(n^2)$ asynchronous rounds. However, these spanning trees are either expensive to maintain or the amount of work in these algorithms is not being considered.

For the case that the network topology is flexible and potentially allows every node to connect to any other node, self-stabilizing algorithms are known that construct a bounded degree spanning tree (e.g., [27]). The algorithm in [27] also has a very low overhead in the stable state. But no formal result is given on the work to establish the spanning tree. Also, an outside rendezvous service, called an oracle, is used to introduce nodes to other nodes, which is not available in our model.

3.4.2 Our Contribution

We show that resource discovery can be solved in $\mathcal{O}(n)$ rounds and that it suffices that each node sends and receives $\mathcal{O}(n)$ messages in total, each message containing $\mathcal{O}(1)$ addresses. We present a protocol P^{CLIQUE} solving the overlay problem $CLIQUE$. The ideas of the protocol P^{CLIQUE} and the results on stabilization time given in Theorem 3.4.8, stabilization work given in Theorem 3.4.13, maintenance work given in Theorem 3.4.14 and the cases of a single joining or leaving node given in Theorem 3.4.15 are based on results published in [38] which is a joint work with my colleague Andreas Koutsopoulos and our supervisor Christian Scheideler:

Sebastian Kniesburges, Andreas Koutsopoulos and Christian Scheideler, A Deterministic Worst-Case Message Complexity Optimal Solution for Resource Discovery, In 20th International Colloquium on Structural Information and Communication Complexity (SIROCCO) 2013 (best student paper).

The results for the correctness in the ATSS model in Theorem 3.4.1 is presented for the first time.

In this section we show that the protocol P^{CLIQUE} solves the clique overlay problem in the ATSS and STSS model with a stabilization time of $\mathcal{O}(n)$ and a stabilization work of $\mathcal{O}(n)$, which is worst-case optimal. We further show a maintenance work per round of $\mathcal{O}(1)$ for each node once a legal state has been reached. We also consider topology updates caused by a single joining or leaving node and show that the network recovers in $\mathcal{O}(n)$ rounds with at most $\mathcal{O}(n)$ messages over all nodes besides the maintenance work.

3.4.3 Formal Definition

According to our model presented in Chapter 2 we assume that each node can be identified by an unique id. We define a hash function $h : ID \mapsto [0, 1]$ that maps the identifier of a node to a point in the $[0, 1]$ interval. We refer to $h(v)$ as v's position. We assume that h is chosen, such that no two nodes get the same position and that h is known to every node, i.e. by knowing the identifier of one node each node can determine its position. Then a global order on the nodes can be defined by their positions $h(v_1) < h(v_2) \cdots < h(v_n)$.

Definition 3.4.1 *A graph $G = (V, E)$ is a* clique*, if $E = \{(u, v) : u \in V \wedge v \in V - \{u\}\}$.*

We now define the internal variables of each node used in our protocol to solve the overlay problem $CLIQUE$. Each node u stores the following variables contributing to the explicit edge set:

- *u.predecessor*: The identifier of u's left neighbor ($h(u.predecessor) < h(u)$) or nil if there is no left neighbor.
- *u.successor*: The identifier of u's right neighbor ($h(u) < h(u.successor)$) or nil if there is no right neighbor.

- *u.neighborhood* The set of nodes u is aware of stored in a cyclic list.
- *u.received-list* The set of nodes u receives by from its predecessor stored in a cyclic list.
- *u.scan-list* The set of nodes u receives by a scan message.

Additionally each node stores the following variables not contributing to the set of explicit edges.

- *u.Ch*: The channel for incoming messages.
- *u.status*: The status of a node, which is by default set to 'inactive' and can be changed to 'active'.
- $u.\tau$: a boolean variable that is periodically true.

To communicate with each other nodes send and receive messages. The content of a message is given by the identifiers of some nodes and a specification of the message type. So different types of messages can trigger different actions. Thus a message m is of the following form $m = (type, ids)$. There are the following types of messages:

- *predecessor-request*: A node u sends this message containing u's identifier to a node v, if u assumes that v is its predecessor.
- *predecessor-accept*: A node u sends this message to a node v containing u's identifier, if u accepts to be v's predecessor.
- *new-predecessor*: A node u sends this message to a node v containing the identifier of a node w, if u rejects to be v's predecessor but recommends w as v's new predecessor.
- *deactivate*: A node u sends this message to a node v containing the identifier of u, to stop v from forwarding messages to u.
- *activate*: A node u sends this message to a node v containing the identifier of u, to allow v to forward messages to u.
- *forward-from-successor*: By this message a node u forwards the identifier of a node w to its predecessor.
- *forward-from-predecessor*: By this message a node u forwards the identifier of a node w to its successor.
- *forward-head*: By this message a node u forwards the identifier of a node w, that has the minimal position of all nodes in u's neighborhood, to its successor.
- *scan*: By this message a node u checks whether the receiving node v and u are in the same spanning tree.
- *scan-acknowledgment*: By this message a node u responds to a scan message and sends the identifier of a node w, that has the minimal position of all nodes in u's neighborhood.
- *delete-successor*: A node u sends this message to a node v if v acted as u's predecessor, but u stores another node as its predecessor.

In the following we define when an assignment of the variables is valid and how the sets of initial topologies IT and the goal topologies $CLIQUE$ look like.

Definition 3.4.2 *An assignment of the variables of a node is* valid *if* $h(u.predecessor) < h(u)$ *with* $u.predecessor \in u.neighborhood$ *or* $u.predecessor = nil$ *and if* $h(u.successor) > h(u)$ *with* $u.successor \in u.neighborhood$ *or* $u.successor = nil$. *If there is some* $v \in u.neighborhood$ *with* $h(v) < h(u)$, *then* $h(u.predecessor) \neq nil$. *If there is some* $v \in u.neighborhood$ *with* $h(v) > h(u)$, *then* $h(u.successor) \neq nil$. *Furthermore* $u.scan - list \subseteq u.neighborhood$ *and* $u.receive - list \subseteq u.neighborhood$. *Note that an invalid assignment can be locally repaired immediately. Thus we assume in the following w.o.l.g. that initially and at every state of the computation the assignment is valid for every node.*

Definition 3.4.3 *Let* E_e *and* E_i *be defined as described in Chapter 2 according to the definition of internal variables. Then the* set of initial topologies *is given by:*

$$IT = \{G = (V, E = E_e \cup E_i) : G \text{ is weakly connected}\}$$

Definition 3.4.4 *Let* E_e *and* E_i *be defined as described in Chapter 2 according to the definition of internal variables. Then the* set of target topologies *is given by:*

$$CLIQUE = \{G = (V, E_e : G \text{ is a clique}\}$$

3.4.4 Protocol P$^{\text{CLIQUE}}$

In order to describe the algorithm formally and prove its correctness later on, we need the definitions given below. We assume that a predecessor of a node is a node with the next smaller position, i.e. for all $u.predecessor$ links, $h(u.predecessor) < h(u)$. Then all nodes in a connected component considering only $u.predecessor$ links form a rooted tree, where for each tree the root has the smallest position.

Definition 3.4.5 *We call such a rooted tree formed by* $u.predecessor$ *links a* heap H. *We further call the root of the tree the* head $h = head(H)$ *of the heap* H. *We further denote with* $heap(u)$ *the heap* H *such that* $u \in H$.

Note here that the *heap* H is not a data structure or variable stored by any node. It is a notion used just for the purpose of the analysis.

Definition 3.4.6 *A heap* H *with head* h, *such that* $\forall v \in H - \{h\} \Rightarrow h(v.predecessor) < h(v)$ *and* $\forall v \in H - \{h\} \Rightarrow v.predecessor.successor = v$ *is a* sorted list. *We call a heap* linearized w.r.t. a node $u \in H$, *if* $\forall v \in H - \{h\} : h(v.predecessor) < h(v)$ *and* $\forall v \in H - \{h\} \wedge h(v) \leq h(u) : v.predecessor.successor = v$. *We further call the time until a heap is linearized w.r.t. a node* u *the* linearization time of u. *We say that two heaps* H_i *and* H_j *are* merged *if all nodes in* H_i *and* H_j *form one heap* H.

Our primary goal is to collect the addresses of all nodes in the system at the node of minimal position, which we also call the *root*. In order to efficiently distribute the addresses from this root to all other nodes in the system, we organize them in a spanning tree of constant degree, which in our case is a sorted list in ascending order of the positions of the nodes. To reach a sorted list, we intermediately organize the

nodes in heaps satisfying the min-heap property, i.e. a *predecessor* of a node has a smaller position than the node itself. The heaps will then be merged and linearized over time so that they ultimately form one sorted list.

We developed this protocol to be worst-case optimal in terms of message complexity in a synchronous setting. Therefore most details of the protocol will come into play when we analyze it in the STSS model. Nevertheless to avoid repetition we give a complete description now.

In our protocol, in order to minimize the amount of messages sent by the nodes, we allow a node u to share information only with $u.successor$ and $u.predecessor$. More precisely a node forwards one of its neighbors (i.e. the nodes it knows about) in a round-robin manner to its predecessor by a forward-from-successor message. The intuition behind this is that if every node does that sufficiently often, eventually the root will learn about all nodes in the system and will send this information back, also in a round-robin manner, to its successor, who will then forward it to its successor by a forward-from-predecessor message until each node is aware of every other node in the system. We illustrate this strategy and our goal topology in Figure 3.15.

Each node periodically computes and updates its successor and predecessor. Each node u chooses the node v in its neighborhood that has the maximal position among all nodes w with a position $h(w) < h(u)$ as its predecessor and requests from it to accept it as its successor by sending a predecessor-request message. Among all nodes that send a predecessor-request message to a node u the one v with the minimal position among all nodes w with a position $h(w) > h(u)$ is selected as $u.successor$ and u sends a predecessor-accept message to v. For all other possible successors this new successor v is recommended as a possible predecessor in a new-predecessor message.

We also need to ensure that there exists a path of successors from the root to all other nodes so that the information of the root reaches every node. Initially there can be many nodes that seem to have the minimal position according to their local knowledge. Each such node is a head of a heap. The challenge is to merge all heaps into one, such that only the root remains as a head. To detect that there are different heaps each head continuously scans its neighborhood.

A node that receives a scan message responds by sending the node with the minimal position in its neighborhood in a scan-acknowledgment message to the node that sent a scan message. In this way a head can decide whether the scanned node belongs to its heap or to another one. If it belongs to another heap the two heaps are merged. To be sure that at some time all nodes in a heap know their head a node forwards it to its predecessor in a forward-head message.

When a node forwards a node by a forward-from-successor resp. forward-from-predecessor message, the node it forwards is the one at the head of the cyclic list $u.neighborhood$ resp. $u.received-list$. Then the head shifts to the next element of the cyclic list. In the same way if a node receives a forward-from-successor resp. forward-from-predecessor message, it stores the received node at the head of its list.

When u has no predecessor that it can send a forward-from-successor message to, although u is not a head, it changes its status to inactive, and then informs its successor by a deactivate message not to send its forward-from-successor messages to u. As soon as u has a predecessor u sends an activate message to $u.successor$ resetting the status to active.

If several nodes store u as their successor, i.e. u receives a deactivate, activate or forward-from-predecessor message by a node that is not its predecessor, then u sends a delete-successor message, correcting the wrong successor link.

An implementation of P^{CLIQUE} in pseudo code can be found in Algorithms 3.4.1- 3.4.16.

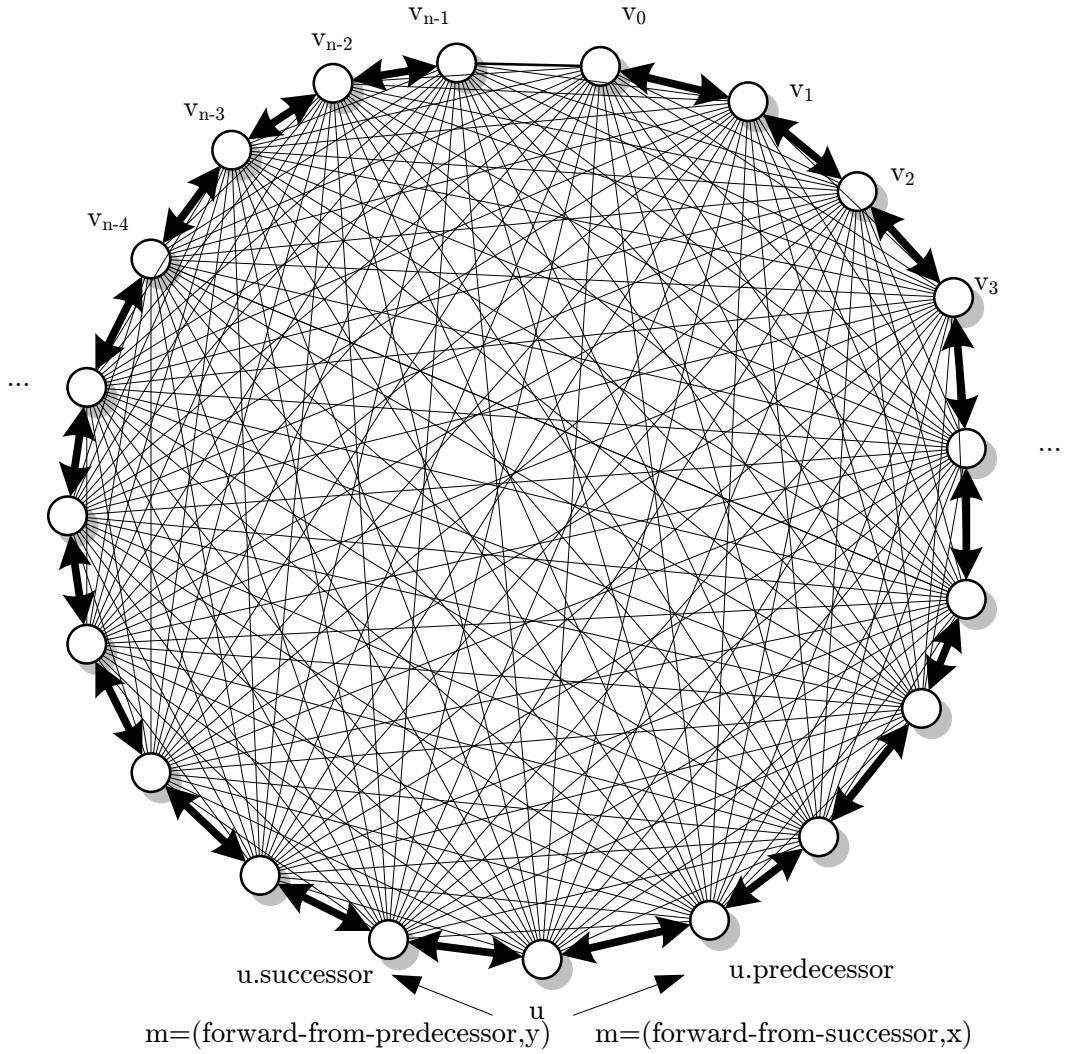


Figure 3.15: The goal topology including a sorted list along which the messages are exchanged

Algorithm 3.4.1 P^{CLIQUE}

message $m = (type, ids) \in u.Ch \rightarrow$
if type=forward-head **then**
  forward-head-message(v_0, v_1)
else if type=scan **then**
  scan-message(v)
else if type=scan-acknowledgment **then**
  scan-acknowledgment-message(v)
else if type=delete-successor **then**
  delete-successor-message(v)
else if type=predecessor-request **then**
  predecessor-request-message(v)
else if type=new-predecessor **then**
  new-predecessor-message(v_0, v_1)
else if type=predecessor-accept **then**
  predecessor-accept-message(v)
else if type=forward-from-predecessor **then**
  forward-from-predecessor-message(v_0, v_1)
else if type=forward-from-successor **then**
  forward-from-successor-message(v_0, v_1)
else if type=deactivate **then**
  deactivate-message(v)
else if type=activate **then**
  activate-message(v)
$\tau \rightarrow$
forward-to-predecessor()
check-head()
forward-to-successor()
forward-head()

Algorithm 3.4.2 forward-to-predecessor()

if $u.status \neq inactive \wedge h(u.predecessor) > nil$ **then** ▷ u is not a head and u is active
  send message $m = (forward-from-successor, u, u.neighborhood[head])$ to $u.predecessor$
▷ forward node to predecessor
  $u.neighborhood[head] = u.neighborhood[head].next$ ▷ shift head to next element in cyclic list

Algorithm 3.4.3 check-head()

if $h(u.predecessor) = nil$ **then** ▷ u is a head, scan a node
  send message $m = (scan, u)$ to $u.neighborhood[head]$
  insert($u.received-list$,$u.neighborhood[head]$,tail) ▷ a copy of$u.neighborhood[head]$ is inserted at the end of $u.received-list$
  $u.neighborhood[head] = u.neighborhood[head].next$ ▷ shift head to next element in cyclic list
else
  send message $m = (predecessor-request, u)$ to $u.predecessor$

Algorithm 3.4.4 forward-to-successor()

if $u.successor \neq nil$ **then**
send message $m = (forward{-}from{-}predecessor, u, u.received{-}list[head])$ to $u.successor$
▷ forward node to successor
$u.received - list[head] = u.received - list[head].next$

Algorithm 3.4.5 forward-head()

$u.neighborhood = u.neighborhood \cup u.scan - list$
$minN =$ argmin$\{h(v) : v \in u.neighborhood\}$
if $h(u.predecessor) \neq nil \wedge u.status \neq inactive$ **then** ▷ forward largest node
send message $m = (forward - head, minN)$ to $u.predecessor$
for all $v \in u.scan - list$ **do** ▷ send the minimum to the nodes of $u.scan - list$
send message $m = (scan - acknowledgment, minN)$ to v
delete($u.scan - list$,v)

Algorithm 3.4.6 forward-head-message(v,)

if $u.status \neq inactive \wedge h(u.predecessor) > nil$ **then** ▷ u is not a head and u is active
if $v \notin u.neighborhood$ **then**
insert(u.neighborhood,v,tail)
if $h(u.predecessor) \neq nil$ **then**
$minN = \{h(w) : w \in u.neighborhood\}$
if $v \neq$ argmin$minN$ **then**
send message $m = (scan - acknowledgment, minN)$ to v
else
send message $m = (scan, u)$ to v

Algorithm 3.4.7 scan-message(v)

if $m.type = scan$ **then** ▷ u has been scanned by a head v
insert($u.scan - list$,v)

Algorithm 3.4.8 scan-acknowledgment-message(v)

if $v \notin u.neighborhood$ **then**
insert(u.neighborhood,v,tail)

Algorithm 3.4.9 delete-successor-message(v)

if $v = u.successor$ **then**
$u.successor = -\infty$
else if $v \notin u.neighborhood$ **then**
insert(u.neighborhood,v,tail)

Algorithm 3.4.10 predecessor-request-message(v)

if $h(v) > h(u)$ **then**
  if $h(u.successor) \neq nil$ **then** ▷ renew successor if necessary, and rearrange old successor
    grandson=argmax$\{h(v), h(u.successor)\}$
    u.successor=argmin$\{h(v), h(u.successor)\}$
    send message $m = (predecessor - accept, u)$ to $u.successor$
    if $u.status \neq inactive$ **then**
      send message $m = (activate, u)$ to $u.successor$
    else
      send message $m = (deactivate, u)$ to $u.successor$
    send message $m = (new - predecessor, u.successor, u)$ to grandson
  else
    u.successor:=v
    send message $m = (predecessor - accept, u)$ to $u.successor$
else if $v \notin u.neighborhood$ **then**
  insert(u.neighborhood,v,tail)

Algorithm 3.4.11 new-predecessor-message(v_0, v_1)

if $v_1 = u.predecessor$ **then**
  if $h(v_0) < h(u) \wedge h(v_0) > h(u.predecessor)$ **then**
    $u.predecessor = v_0$
    send message $m = (predecessor - request, u)$ to $u.predecessor$
    status(u)=inactive
    if $u.successor \neq nil$ **then**
      send message $m = (deactivate, u)$ to $u.successor$
  else if $v_0 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v,tail)
else
  if $v_0 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v,tail)
  if $v_1 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v,tail)

Algorithm 3.4.12 predecessor-accept-message(v)

if $v \neq u.predecessor$ **then** ▷ the predecessor has accepted u as its successor
  if $v \notin u.neighborhood$ **then**
    insert(u.neighborhood,v,tail)
  send message $m = (delete - successor, u)$ to v

Algorithm 3.4.13 deactivate-message(v)

if $v = u.predecessor$ **then**
  status(u):=inactive
  if $u.successor \neq nil$ **then**
    send message $m = (deactivate, u)$ to $u.successor$ ▷ forward the deactivation message to successor
else
  send message $m = (delete - successor, u)$ to v
  if $v \notin u.neighborhood$ **then**
    insert(u.neighborhood,v,tail)

Algorithm 3.4.14 activate-message(v)

if $v = u.predecessor$ **then**
  status(u):=active
  if $u.successor \neq nil$ **then**
    send message $m = (activate, u)$ to $u.successor$ ▷ forward the activation message to successor
else
  send message $m = (delete - successor, u)$ to v
  if $v \notin u.neighborhood$ **then**
    insert(u.neighborhood,v,tail)

Algorithm 3.4.15 forward-from-successor-message(v_0, v_1)

if $v_1 = u.successor$ **then** ▷ insert the node forwarded from $u.successor$ to $u.neighborhood$
  insert(u.neighborhood, v_0, head)
else
  if $v_1 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v_1,tail)
  if $v_0 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v_0,tail)

Algorithm 3.4.16 forward-from-predecessor-message(v_0, v_1)

if $v_1 = u.predecessor$ **then** ▷ insert the node forwarded from $u.predecessor$ to $u.neighborhood, u.received - list$
  insert(u.neighborhood, v_0, tail)
  insert(u.received-list, v_0, head)
else
  if $v_1 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v_1,tail)
  if $v_0 \notin u.neighborhood$ **then**
    insert(u.neighborhood,v_0,tail)
  send message $m = (delete - successor, u)$ to v_1

3.4.5 Analysis in the ATSS model

We now show the correctness of our approach in the ATSS model by proving that it stabilizes to a clique. We show this by dividing the self-stabilization process into different phases and determine the correctness of each phase. In particular we will show that if each node executes the protocol P^{CLIQUE} started in an weakly connected graph, eventually a graph is reached, such that the sorted list is sub-graph. We then show that if started in a graph that contains the sorted list as a sub-graph P^{CLIQUE} leads to clique. In fact we show the following main theorem:

Theorem 3.4.1 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then eventually the graph converges to a graph $G' \in CLIQUE$ (Convergence). If in an initial graph $G \in CLIQUE$ every node executes the protocol P^{CLIQUE} then each possible computation leads to a graph $G' \in CLIQUE$ (Closure).*

To prove this theorem we introduce some formal definitions of the intermediate graphs and subsets of the edges.

Definition 3.4.7 *We define:*

- $E_{e-heap} = \{(x, y) \in E_e : y = x.predecessor\} \subseteq E_e$ *is the subset of explicit edges that will form a heap.*
- $E_{e-list} = \{(x, y) \in E_e : y = x.predecessor \vee y = x.successor\} \subseteq E_e$ *is the subset of explicit edges that will form the sorted list.*
- $E_{i-list} = \{(x, y) \in E_i : \exists m = (list, y) \in x.Ch\}$ *is the subset of implicit edges that are needed to form the sorted list.*
- $E_{list} = E_{e-list} \cup E_{i-list}$ *is the set of edges that take part in the process forming the sorted list.*

Definition 3.4.8 *A graph $G = (V, E)$ is a* sorted list*, if $V = v_0, \cdots v_{n-1}$ with $h(v_i) < h(v_i + 1) \forall i \in \{0, \cdots n-1\}$ and $E = \{(v_i, v_j) :, j = i - 1 \vee j = i + 1 \forall 0 < j < n - 1\}$.*

Convergence

We start by proving the convergence property of P^{CLIQUE}.

Theorem 3.4.2 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then eventually the graph converges to a graph $G' \in CLIQUE$.*

To show this we will prove a set of intermediate theorems first, from which the convergence follows. Our first theorem claims that the computation reaches a state, such that the sorted list is a sub-graph of the graph of explicit edges.

Theorem 3.4.3 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then eventually the computation reaches a state s_t such that G^t_{e-heap} is one connected component, i.e. weakly connected, in every state s'_t, $t' > t$.*

Proof. Obviously if G^t is weakly connected, then G^{t+1} is weakly connected, as in P^{CLIQUE} nodes in $u.neighborhood$ are never deleted and received identifiers are stored in $u.neighborhood$, if they are not already in $u.neighborhood$.

After establishing that connectivity is preserved, we define a time t_0 at which every message initially in $u.Ch$ for any node u has been received, the messages in response have been delivered, each node executed the periodic actions at least once and the corresponding messages and the responses have been received. Then we know that every message $m = (new - predecessor, w, u)$ sent by a node u at time $t_i \geq t_0$ contains a node w with $w = u.successor$ and $u = w.predecessor$.

Then G^{t_0} consists of several heaps $H_1, H_2 \cdots H_k$. In the following we show that eventually all these heaps are merged to one single heap as illustrated in Figure 3.16.

If two nodes u and v are in the same heap H_i at time $t > t_0$ they will be in the same heap at time $t + 1$. Heaps are formed by edges $(u, v) \in E^t_{e-heap}$ with $v = u.predecessor$. Thus it suffices to show that a pair u, v with $v = u.predecessor$ stays in the same heap. u and v can only be in different heaps if $u.predecessor$ is changed at time $t + 1$. According to the protocol P^{CLIQUE} there are only two cases in which $u.predecessor$ is changed. If $u.predecessor = nil$ and u receives a node w with $h(w) < h(u)$, or if u receives a new-predecessor message $m = (new-predecessor, w, v = u.predecessor)$. Then we can follow from our observation that $w = v.successor$ and $w.predecessor = v = u.predecessor$. Thus if u updates $u.predecessor$ to w $(u, w = u.predecessor) \in E^{t+1}_{e-heap}$ and $(w, v = w.predecessor) \in E^t_{e-heap}$ and thus u and v are still in the same heap. If $v \neq w.predecessor$ at time $t + 1$ then w received a new-predecessor message from v before and the argument can be applied inductively.

As G^0 is weakly connected and stays weakly connected there is also a connection between all heaps $H_1, H_2 \cdots H_k$ at time t_0. Let $(u, v) \in E^{t_0}$ with $u \in H_i$ and $v \in H_j$ be an edge connecting two heaps. Then there is a state s_{t_1} with $(u, v) \in E^{t_1}_e$ and $v \in u.neighborhood$. Then according to P^{CLIQUE} eventually u will send v to its predecessor. By applying this argument inductively we can show that the computation reaches a state t_2 such that $(u', v) \in E^{t_2}$ and $u' = head(heap(u))$. Then if $h(v) < h(u')$, u' will set $u'.predecessor$ to v and u' and v are now in the same heap. If $h(v) > h(u')$ u' will eventually send a scan message to v. Now there can be three cases:

- in $v.neighborhood$ is no node q with $h(q) < h(u')$ and $v.predecessor = nil$, then v sets $v.predecessor = u'$.
- in $v.neighborhood$ is no node q with $h(q) < h(u')$ then v will forward u' to its predecessor v', then for v' there can be the same three cases. Thus eventually case 1 or three holds.
- in $v.neighborhood$ is a node q with $h(q) < h(u')$, then v will send a scan-acknowledgment message to q containing w. By the same arguments as for the edge (u, v) we can also show that eventually the computation reaches a state such that $(v', q) \in E^t$ and $v' = head(heap(v))$. Then to show that u and v are in the same heap it suffices to show that v' and q and u' and q are in the same heap. As $h(v') < h(v)$, $h(u') < h(u)$ and $h(q) < h(u)$ and $h(q) < h(v)$ this induction terminates and u and v are in the same heap.

Then we can conclude immediately that eventually all nodes are in one single heap. □

Theorem 3.4.4 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then eventually the computation reaches a state s_t such that G^t_{e-list} is a sorted list.*

Proof. According to Theorem 3.4.3 the computation contains a state s_{t_0} such that all nodes form one heap in every state s_t, $t \geq t_0$. Let $v_0, v_1, \cdots v_{n-1}$ be the nodes in sorted order such that $h(v_i) < h(v_i+1)$.

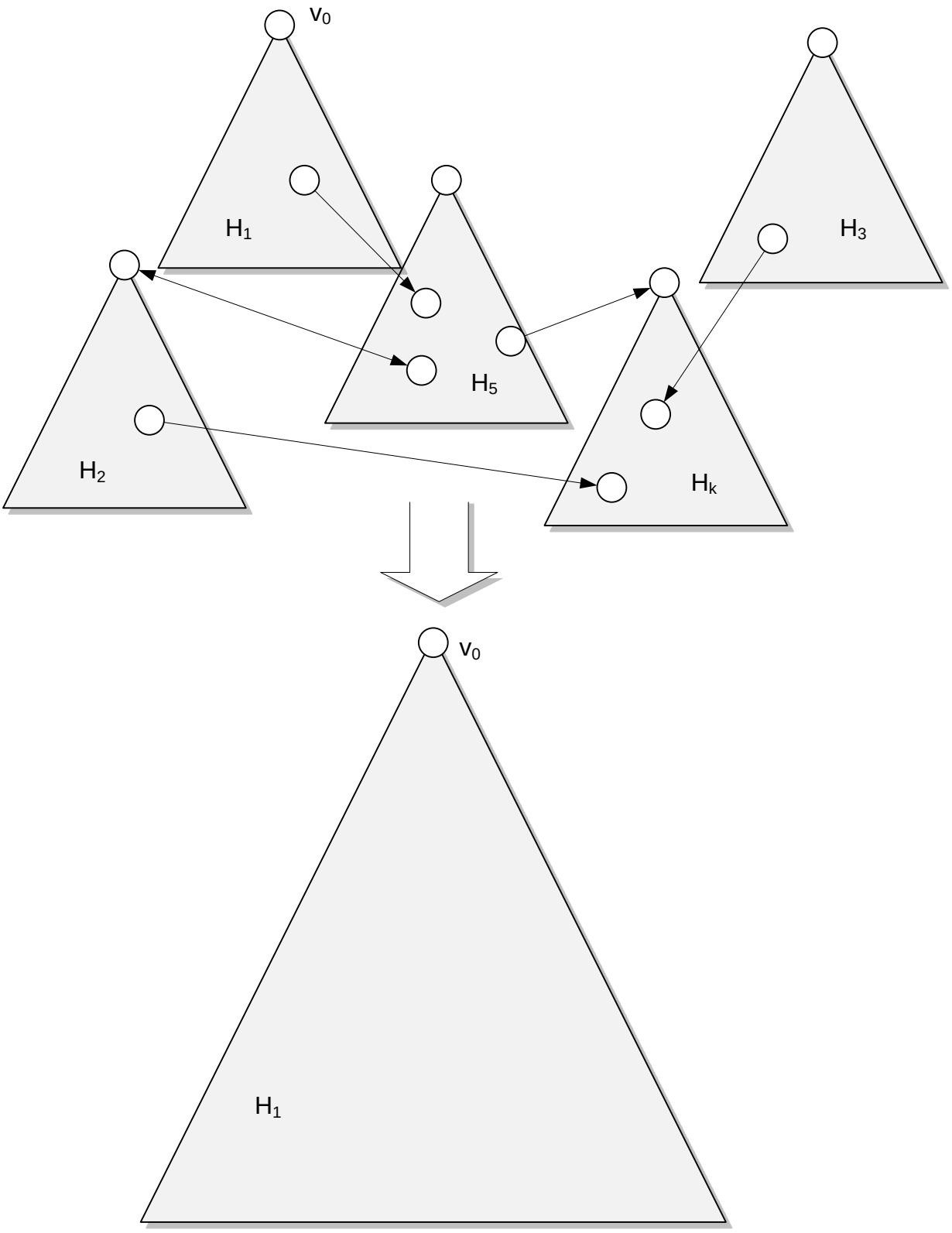


Figure 3.16: All single heaps are merged into one

We then show by induction that the computation contains a state s_{t_i} such that the heap is linearized w.r.t. v_i. Then all nodes will eventually form a sorted list as illustrated in Figure 3.17.

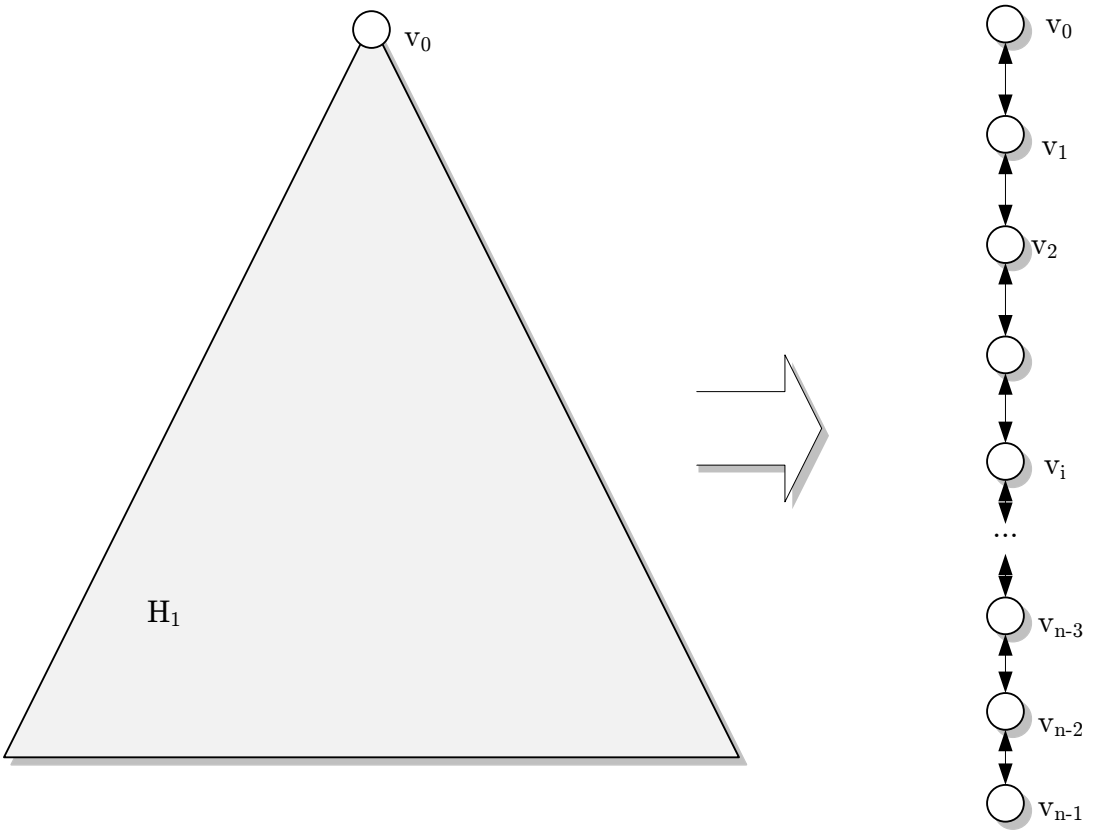


Figure 3.17: A single heap is linearized to one sorted list

Obviously this holds for t_0. Let's assume the hypothesis holds for t_i, then there is a state $s_{t_{i+1}}$ at time $t_{i+1} > t_i$ when every node has sent its predecessor-request message and received a corresponding predecessor accept or new-predecessor message. If there are several nodes w with $w.predecessor = v_i$ then all but v_{i+1} will receive a new-predecessor message thus for all $w \neq v_{i+1}$ $w.predecessor \neq v_i$. Furthermore v_i sets $v_i.successor = v_{i+1}$. Thus the heap is linearized w.r.t. v_{i+1} at this state $s_{t_{i+1}}$. □

Theorem 3.4.5 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then eventually the computation reaches a state s_t such that G^t_{e-list} is a sorted list and for the root w with $w = head(heap(v))\forall v \in V$ and $u.neighborhood = V - \{w\}$.*

Proof. According to Theorem 3.4.4 the computation contains a state s_{t_0} such that all nodes form a sorted list in every state s_t, $t \geq t_0$. Let $v_0, v_1, \cdots v_n$ be the nodes in sorted order such that $h(v_i) < h(v_i + 1)$. Then according to P^{CLIQUE} all nodes will forward their knowledge to their predecessors, whereby new information is forwarded with a higher priority as it is inserted at the head of $u.neighborhood$, thus eventually all information is accumulated at the root v_0. □

Theorem 3.4.6 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then eventually the computation reaches a state s_t such that G_e^t is a clique.*

Proof. Let $v_0, v_1, \cdots v_n$ be the nodes in sorted order such that $h(v_i) < h(v_i + 1)$. Then according to Theorem 3.4.5 the computation contains a state s_{t_0} such that all nodes form a sorted list and $v_0.neighborhood = V - v_0$ in every state s_t, $t \geq t_0$.Then according to P^{CLIQUE} the root v_0 will forward its information in round-robin manner to its successor. In the same way each node forwards the information received by the predecessor to its successor. Thus eventually the information sent by the root reaches every node and thus $u.neighborhood = V - \{u\}\ \forall u \in V$. Then G_e^t is a clique, as $E_e^t = \{(u, v) : u \in V \wedge v \in V - \{u\}\}$. □

Closure

We will now show that for P^{CLIQUE} also the closure property holds.

Theorem 3.4.7 *If in an initial graph $G \in CLIQUE$ every node executes the protocol P^{CLIQUE} then each possible computation leads to a graph $G' \in CLIQUE$.*

Proof. As G_e^t is a clique and thus $u.neighborhood = V - \{u\}\ \forall u \in V$ and no nodes are deleted from $u.neighborhood$ according to P^{CLIQUE}, also G_e^{t+1} is a clique. □

3.4.6 Analysis in the STSS model

In this section we analyze P^{CLIQUE} according to the STSS model. In particular we consider the stabilization time, stabilization work and the maintenance work.

Stabilization time

Theorem 3.4.8 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol P^{CLIQUE} then the graph converges to a graph $G' \in CLIQUE$ (Convergence) with a stabilization time of $\Theta(n)$.*

For our analysis we introduce some additional definition.

Definition 3.4.9 *We define an* undirected path *as a sequence of edges $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$, such that $\forall i \in \{1, \cdots, k\} : (v_i, v_{i-1}) \in E \vee (v_{i-1}, v_i) \in E$.*

Definition 3.4.10 *We introduce a further edge set $E_{scan} E_{e-scan}^t \cup E_{i-scan}^t$ of edges resulting from scan-messages with $E_{e-scan}^t = \left\{(u, v) \in E^t : v = \text{argmin}\{h(w) : w \in u.scan - list\}\right\}$ and $E_{i-scan}^t = \left\{(u, v) \in E^t : m = (forward - head, v) \in u.Ch\right\}$. We call edges in E_{scan}* s-edges.

Definition 3.4.11 *We say that two heaps H_1 and H_2 are* s-connected *if there exists at least one undirected path from one node in H_1 to one node in H_2 and this path consists of either s-edges or edges having both nodes in the same heap.*

Like in the analysis of P^{CLIQUE} in the ATSS model we will split the proof in some intermediate steps. First we show that starting from a valid state all existing heaps will eventually be connected by s-edges, so that they can merge afterward.

Theorem 3.4.9 *If H_i and H_j are two heaps connected by $(u, v) \in E^t$ then after $\Theta(n)$ rounds the heaps have either merged or are connected by s-edges.*

Before we show the theorem we prove some helpful lemmas.

Lemma 3.4.1 *Let $v_0, \cdots v_{|H|-1}$ be the elements in a heap H in ascending order of their position, i.e $h(v_i) < h(v_{i+1})$. Then it takes at most i rounds until H is linearized w.r.t v_{i-1}.*

Proof. We prove the lemma by induction on the number of rounds i. Note that all nodes are connected by the $u.predecessor$ links only to nodes with smaller positions.

Induction base ($i = 0$): The head of the heap is the node with the minimal position v_0 therefore trivially, $\forall v \in H - \{head(H)\} : h(v) > h(v.predecessor)$ and $\forall v \in H - \{head(H)\} \Rightarrow v.predecessor.successor = v$.

Induction step ($i \rightarrow$ i+1): By induction the heap is linearized w.r.t. v_{i-1} after at most i rounds, thus v_i has to be connected to v_{i-1} by a $v.predecessor$ link. In the $i+1$th round v_{i-1} sends $new-predecessor$ messages to all other nodes with $v.predecessor = v_{i-1}$, such that $v_{i-1}.successor = v_i$ and v_i becomes the only node with $v_i.predecessor = v_{i-1}$. Then $\forall v \in H - \{h\} : h(v) > h(v.predecessor)$ and $\forall v \in H - \{h\} \Rightarrow v.predecessor.successor = v$. □

Lemma 3.4.2 *If two heaps H_i and H_j merge to one heap H, the linearization time of a node $u \in H_i$ (resp. $u \in H_j$) can increase by at most $|H_j|$ (resp. $|H_i|$).*

Proof. Without loss of generality let $u \in H_i$. By Lemma 3.4.1 we know that the linearization time depends on the number of nodes with a smaller position in the heap. The number of nodes with a smaller position can increase by at most the size of the other heap H_j. Thus, also the linearization time can only increase by at most $|H_j|$. □

We introduce some additional notation to estimate the time it takes until some node is scanned by a head of a heap.

For any edge $(u, v) \in E$ with $u \in H_i$ and $v \in H_j$, where h_i and h_j denote the corresponding heads of the heaps, we define the following notation for the tth round: Let $P^t(u)$ be the *length of the path* from u to h_i, once H_i is linearized w.r.t. u. Let $ID^t(u, v)$ be the *number of ids* u forwards or scans before sending or scanning v the first time. Let $LT^t(u)$ be the *time* it takes until the heap is *linearized* w.r.t. u , i.e. on the path from the head h_i to u each node has exactly one predecessor and successor. Lemma 3.4.1 shows that $LT^t(u)$ is bounded by $|H_i|$.

Let $\phi^t(u, v) = P^t(u) + ID^t(u, v) + LT^t(u)$. We call $\phi^t(u, v)$ the *delivery time* for a node v because if $\phi^t(u, v) = 0$, v is scanned in round t or has already been scanned by h_i. We then denote by $\Phi^t(u, v) = \min\left\{\phi^t(w, v) : heap(u) = heap(w)\right\}$ the minimal delivery time of v for any node in the same heap as u.

For any edge $(u, v) \in E$, with $u \in H_i$ and $v \in H_j$, (i.e. u and v are in different heaps) and $\Phi^t(u, v) = 0$ the head of H_i scans or has scanned $v \in H_j$ resulting in the s-edge $(v, h_i)_s$.

We are now ready to prove Theorem 3.4.9

Proof. We start by showing an upper bound of $\mathcal{O}(n)$ rounds. It takes at most 2 rounds until each node has received the messages initially in the system at time $t = 0$ and the corresponding responses. We call this time t_0. After t_0 a node u only receives a forward-from-successor message at time $t \geq t_0$ from a node v with $v = u.successor$ at time $t - 1$. A node only sends a forward-from-successor message

if its predecessor accepted it as the successor. Thus as soon as one node in a round $t > 2$ forwards an identifier w to its predecessor by a forward-from-successor message, each node u receiving this message either forwards it to its predecessor immediately or is inactive and forwards w the next time it is active. Note that an inactive node receives at most one forward-from-successor message while being inactive, as also the successor will be deactivated in the next round. Thus $w = u.neighborhood[head]$ while u is inactive.

We claim that, if $(u, v) \in E^{t_0}$ is an edge between two heaps H_i and H_j, then for all rounds $t \geq t_0$:

$$\Phi^t(u, v) \leq \max\{2|H_i| + n - (t - t_0), 0\} \leq \max\{3n - (t - t_0), 0\}$$

We will show this claim by induction on the number of rounds. For the analysis we divide each round $t \to t + 1$ into two parts: in the first step $t \to t'$ all actions are executed and in the second step $t' \to t + 1$ all network changes are considered. Thus, we assume that all actions are performed before the network changes. This is reasonable as a node is aware of changes in its neighborhood only in the next round, when receiving the messages. By network changes we mean the new edges that could be created in the network. These new edges could possibly lead to the merging of some heaps at time $t + 1$.

Induction base($t = t_0$):

For any edge $(u, v) \in E^{t_0}$ between H_i and H_j let $u \in H_i$ be the node such that $\Phi^0(u, v) = \phi^0(u, v)$. Then $P^0(u) \leq H_i$ as the path length is limited by the number of nodes in the heap, $ID^0(u, v) \leq n$ as not more than n ids are in the system, and following from Lemma 3.4.1, $LT(u) \leq |H_i|$. Then $\Phi^0(u, v) \leq \phi^0(u, v) \leq 2|H_i| + n \leq 3n$.

Induction step($t \to t'$): For any edge $(u, v) \in E^{t_0}$ between H_i and H_j let $u \in H_i$ be the node such that $\Phi^t(u, v) = \phi^t(u, v)$.

Then in round t the following actions can be executed.

- u is inactive and can not forward an id. Then the heap is not linearized w.r.t. u, which implies that the linearization time decreases by one, i.e. $LT^{t'}(u) = LT^t(u) - 1$ and $\phi^{t'}(u, v) = \phi^t(u, v) - 1 \leq 2|H_i| + n - (t - t_0) - 1$ as all other values are not affected.

- u is active, but does not send v by a *forward-from-successor* message, then the number of ids that u is sending before v decreases by 1. Note that u hasn't sent a *forward-from-successor* message with v in a round before, as then according to our observation above there would be another node $y \in H_i$ with $\phi^t(y, v) < \phi^t(u, v)$. Then $ID^{t'}(u, v) \leq ID^t(u, v) - 1$ and $\phi^{t'}(u, v)) = \phi^t(u, v) - 1 \leq 2|H_i| + n - (t - t_0) - 1$.

- u sends a *forward-from-successor* message with v, then the length of the path for v to the head h_i decreases by 1 and $\phi^{t+1}(u.predecessor, v) \leq P^t(u) - 1 + ID^t(u, v) + LT^t(u) = \phi^t(u, v) - 1 \leq 2|H_i| + n - (t - t_0) - 1$

Thus, in total $\Phi^{t'}(u, v) \leq \Phi^t(u, v) - 1 \leq 2|H_i| + n - (t - t_0) - 1 \leq 3n - ((t - t_0) + 1)$.

Induction step($t' \to t + 1$): Now we consider the possible network changes and their effects on the potential $\Phi^{t+1}(u, v)$. Let again $u \in H_i$ be the node such that $\Phi^t(u, v) = \phi^t(u, v)$ for an edge $(u, v) \in E^{t_0}$ between H_i and H_j. The following network changes might occur:

- some heaps H_k and H_l with $k \neq i$ and $l \neq i$ merge. This has no effect on $\Phi^{t'}(u, v)$. Thus, $\Phi^{t+1}(u, v) = \Phi^{t'}(u, v) \leq 2|H_i| + n - (t - t_0) - 1 \leq 3n - (t + 1 - t_0)$.

- Heaps H_i and H_k merge to H_i'. Obviously the length of the path of u can increase and $P^{t+1}(u) \leq P^{t'}(u) + |H_k|$. According to Lemma 3.4.2 also the linearization time of u can increase and $LT^{t+1}(u) \leq LT^{t'}(u) + |H_k|$. In total $\Phi^{t+1}(u,v) \leq \Phi^{t'}(u,v) + 2|H_k| \leq 2|H_i'| + n - (t - t_0) - 1 \leq 3n - (t + 1 - t_0)$.

Thus, in round $t+1$, $\Phi^{t+1}(u,v) \leq 2|H_i| + n - (t - t_0) - 1 \leq 3n - (t + 1 - t_0)$. Hence for every edge $(u,v) \in E_0^t$with $u \in H_i$ and $v \in H_j$, $\Phi^t(u,v) = 0$ after $3n$ rounds after t_0, which means that the head of H_i scans or has scanned $v \in H_j$ resulting in a s-edge (v, h_i).

It remains to show that two heaps connected by s-edges stay connected. If there is a s-edge (u,v) connecting two heaps H_i and H_j at time t then if u is not the head of H_i u either forwards v to its predecessor and H_i and H_j stay connected by the s-edge $(u.predecessor, v)$ or u sends a scan-acknowledgment to v with $w = \text{argmin}\,\{h(w') : w' \in u.neighborhood\}$, then H_i and H_j stay connected by the s-edges (u,w) and (v,w). If u is the head of H_i then either v becomes u' predecessor and the heaps are merged if $h(v) < h(u)$ or u sends a scan message containing u itself and H_i and H_j are connected by the s-edge (v,u).

Obviously $\Omega(n)$ is a lower bound for the stabilization time. Say a node u is connected to $\Omega(n)$ nodes including a node v and the edge (u,v) is the only edge connecting v to the rest of the network. Then if u stores v in the last position of $u.neighborhood$ u forwards or scans $\Omega(n)$ nodes before v. Thus the stabilization time is bounded by $\Omega(n)$. □

Based on the results of Theorem 3.4.9, we will prove that after $O(n)$ further rounds all nodes form one heap.

Theorem 3.4.10 *If all heaps are connected by s-edges after $\mathcal{O}(n)$ rounds all nodes form one single heap.*

Proof. Let $v_0, v_1 \cdots v_n$ be sorted by the position in ascending order, such that $h(v_i) < h(v_{i+1})$. We define the following potential for a s-edge (v_i, v_j) as $\sigma(v_i, v_j) = 2 \cdot i + 2 \cdot j + I(v_i, v_j)$, where $I(v_i, v_j) = 1$ if $j > i$ and 0 otherwise. If v_i and v_j are in the same heap $\sigma(v_i, v_j) = 0$. We proceed by showing that in each round a s-edge is substituted by a set of s-edges with a smaller potential or the connected heaps are merged. Let $(v_i, v_j)_s$ be a s-edge connecting two heaps H_l and H_k, i.e. $v_i \in H_l$, $v_j \in H_k$. Then according to our algorithm the following actions might be executed.

- v_i is the head of H_l and $i > j$ then $v_j = v_i.predecessor$ and v_i sends a predecessor-request message to v_j, resulting in a merge of H_l and H_k.
- v_i is the head of H_l and $j > i$ then v_i sends a scan message to v_j with its own identifier and the edge $(v_j, v_i)_s$ is created connecting H_l and H_k. Then $\sigma(v_j, v_i) = 2i + 2j + 0 < 2i + 2j + 1 = \sigma(v_i, v_j)$.
- v_i forwards v_j to $v_m = v_i.predecessor$ by a forward-head message, such that H_l and H_k are connected by $(v_i.predecessor, v_j)_s$. Then $\sigma(v_i.predecessor, v_j) = 2m + 2j + I(v_m, v_j) < 2i + 2j + I(v_i, v_j) = \sigma(v_i, v_j)$ as $m < i$.
- v_i sends $v_m = \text{argmin}\,\{h(w) : w \in v_i.neighborhood\}$ in a scan-acknowledgment message to v_j with $m < i$ and $m < j$ and the s-edge $(v_i, v_j)_s$ is substituted by s-edges $(v_i, v_m)_s$ and $(v_j, v_m)_s$. And H_l and H_k are connected via s-edges. The potential of the new edges is:
  - $\sigma(v_i, v_m) = 2i + 2m + I(v_i, v_m) < 2i + 2j + I(v_i, v_j) = \sigma(v_i, v_j)$
  - $\sigma(v_j, v_m) = 2j + 2m + I(v_i, v_m) < 2i + 2j + I(v_i, v_j) = \sigma(v_i, v_j)$

The maximal potential is bounded by $\sigma(u, v) < 4n + 1$. By Theorem 3.4.9 we know that there is a state at a time t_1, such that all heaps are connected by s-edges. Thus after $\mathcal{O}(n)$ rounds each pair of heaps that was connected at time t_1 by s-edges is connected by a sequence of s-edges with potential 0. Then all nodes form one heap. □

Theorem 3.4.11 *If all nodes form one heap, it takes $\mathcal{O}(n)$ rounds until the computation reaches a state s_t such that $G^t \in CLIQUE$ is a clique.*

Proof. Since at this point we only have one head the heap will be linearized after $\mathcal{O}(n)$ rounds. This follows directly from Lemma 3.4.1. Once the heap is linearized and forms a sorted list, each node will be sent to the root, the remaining head, after at most $2n$ rounds. So the root will be aware of every node. The root has sent all nodes in its neighborhood to its successor after n rounds. The successor of the root forwards the received information to its successor. As a consequence, all nodes will receive all nodes after n further rounds. Adding all this together, after $O(n)$ rounds all nodes will know each other and a clique will be constructed. □

Combining Theorem 3.4.9, Theorem 3.4.10 and Theorem 3.4.11 our main theorem Theorem 3.4.8 holds and P^{CLIQUE} needs a stabilization time of $\Theta(n)$.

We conclude the analysis by showing the Closure property in the STSS model.

Theorem 3.4.12 *If in an initial graph $G \in CLIQUE$ every node executes the protocol P^{LIST} then each possible computation leads to a graph $G' \in CLIQUE$ (Closure).*

Proof. The proof is the same as in the asynchronous setting. □

Stabilization work

According to Theorem 3.4.8 it takes $\Theta(n)$ rounds to reach a legal state. In each round $t > 0$ each active node sends a message to its predecessor and its successor (forward-from-successor, forward-from-predecessor) and receives a message from them (forward-from-successor, forward-from-predecessor). Also, a node sends at most one activate/deactivate message to its successor at each round. This gives a resulting work of $\Theta(n)$ for each node. By the following lemmas we show that the additional messages sent and received during the stabilization are at most $\mathcal{O}(n)$ for each node.

Lemma 3.4.3 *Each node sends and receives at most $\mathcal{O}(n)$ predecessor-request, predecessor-accept and new-predecessor messages during the stabilization process.*

Proof. In each round $t > t_0$ each node sends at most one predecessor-request and one predecessor-accept message and receives at most one predecessor-accept or new-predecessor message. It remains to show that each node receives at most $\mathcal{O}(n)$ predecessor-request messages and sends at most $\mathcal{O}(n)$ new-predecessor messages. Note that it suffices to show that each node receives at most $\mathcal{O}(n)$ predecessor-request, as the number of new-predecessor messages directly depends on the number of received predecessor-request messages. To each node that sends a predecessor-request to u that is not u' successor, u sends a new-predecessor message. A node u only sends at most one new-predecessor message to each other node v. By receiving this message v changes its predecessor. Thus before u sends another new-predecessor message to v, v has to change its predecessor back to u. A predecessor is only changed if a root receives a node with a smaller position, or if the predecessor of a node sends a new-predecessor. v cannot be a head,

thus v's predecessor is only changed by another new-predecessor message. But v's predecessor can not be changed back to u as the position of the new predecessor is strictly increasing. By this monotonicity it follows that a node u only sends at most one new-predecessor message to each other node v. Thus, every node only sends and receives $\mathcal{O}(n)$ predecessor-request and new predecessor messages. □

Lemma 3.4.4 *Each node sends and receives at most $\mathcal{O}(n)$ scan and scan-acknowledgment messages during the stabilization.*

Proof. Only heads of heaps send scan messages. In each round each head sends $\mathcal{O}(1)$ scan messages. One in the periodic actions and another one if a head receives a forward-head message with a node with a greater position. Each scanned node sends a scan-acknowledgment message back or stores the sending node in $u.scan-list$. Obviously a node can be scanned by up to n different heads in one round. Which would lead to a work of $\mathcal{O}(n^2)$ by receiving these messages. But as a node sends the node with minimal position in its neighborhood with a scan-acknowledgment message, it is scanned at most once by heads with a greater position. By receiving this node with minimal position the scanning node recognizes, if it is still a head, that it cannot be a head of the heap and sets its predecessor and stops scanning. So a node can be scanned by $\mathcal{O}(n)$ heads. Regarding the scan-acknowledgment messages, since each head sends $\mathcal{O}(1)$ scan messages in each round, it receives also at most $\mathcal{O}(1)$ scan-acknowledgment messages in each round. So, all in all, a node receives $\mathcal{O}(n)$ messages during the stabilization. □

Lemma 3.4.5 *Each node sends and receives at most $\mathcal{O}(n)$* forward-head *messages through the stabilization phase.*

Proof. A node sends at most one forward-head message per round $t > t_0$. The number of forward-head messages it receives during the stabilization is limited by $\mathcal{O}(n)$. That is because each node u receives one forward-head message from its successor in a round, and possibly from other possible successors, let v be such one, for which $v.predecessor = u$. But v can only be once a possible successor of u, since at the next round it either will be forwarded to $u.successor$ and will never have u as its predecessor again, or it becomes $u.successor$. Since each node can be only once a possible successor for u, the number of forward-head messages sent through all possible successors is limited by n. So, the number of forward-head messages it receives during the stabilization is limited by $\mathcal{O}(n)$. □

Combining Lemmas 3.4.3,3.4.5 and 3.4.5 we get the following theorem:

Theorem 3.4.13 *If in an initial graph $G \in IT$ every node executes the protocol P^{CLIQUE} then each possible computation leads to a graph $G^{CLIQUE} \in CLIQUE$ with a stabilization work of $\Theta(n)$.*

Maintenance work

Theorem 3.4.14 *If in an initial graph $G \in CLIQUE*$ every node executes the protocol P^{CLIQUE} then each possible computation leads to a graph $G' \in CLIQUE*$ with a maintenance work of $\mathcal{O}(1)$.*

Proof. In a legal state $G \in CLIQUE*$ all nodes form a sorted list. Thus, each node has exactly one stable successor and one stable predecessor. Then each node sends and receives one predecessor-request and one predecessor-accept message. Each node sends one forward-from-successor and one forward-from-predecessor message. Moreover there is one head that sends one scan message, which is received by one other node, and receives one scan-acknowledgment, sent by the scanned node. Thus, each node sends and receives $\mathcal{O}(1)$ messages in a stable state. □

Single Join and Leave Event

Theorem 3.4.15 *In a legal state it takes $\mathcal{O}(n)$ rounds and messages to recover and stabilize after a new node joins the network. It takes $\mathcal{O}(1)$ rounds and messages to recover the clique after a node leaves the network.*

Proof. If a node u joins the network it creates an edge (u, v) to a node v in the clique. If $h(v) < h(u)$, u sends a predecessor-request to v, v then either accepts u as its successor or creates an edge from u to v's successor. It takes at most $\mathcal{O}(n)$ rounds until v reaches its final position in the sorted list. Additionally v sends u to its predecessor, and after $\mathcal{O}(n)$ rounds the root inserts u to its neighborhood. After $\mathcal{O}(n)$ further rounds each node receives u's identifier and u receives the identifier of all other nodes in the network.

If $h(v) > h(u)$ u will scan v. Then v sends u's identifier to its predecessor, because it is a new node in its neighborhood. If $h(u) > h(root)$ v will respond by a scan-acknowledgment containing the root. Then u assumes the root to be its predecessor and u is connected to a node $v' = root$ with $h(u) > h(v')$ and the analysis from above holds.

If $h(u) < h(root)$ then v will forward u by a forward-head message. After $\mathcal{O}(n)$ rounds the root receives u and will set u as its new predecessor. Then by the following forward-from-successor messages u will receive the identifier of any other node in $\mathcal{O}(n)$ rounds. After $\mathcal{O}(n)$ further rounds each node receives u's id. Thus, after $\mathcal{O}(n)$ rounds after a join the nodes form a clique and the sorted list is formed.

As each node only receives and sends $\mathcal{O}(1)$ messages in each round it also takes $\mathcal{O}(n)$ messages to form a clique after one single node joins a stable clique.

Obviously a clique remains a clique in case a node u leaves the network. Also the sorted list is immediately repaired, as the successor of the removed node, assumes u' predecessor to be its predecessor and sends a predecessor-request, which will be accepted as the node has no other successor. Note that if u is the root of the list, u's successor will recognize that there is no node with a smaller position in its neighborhood and will correctly assume to be a head of a list and proceed the scanning. □

3.5 Conclusion & Outlook

In this chapter we considered self-stabilizing protocols for three different overlay networks, namely a sorted list, a specific small-world network and a clique. We analyzed the protocols according to the ATSS and STSS model. For the P^{LIST} protocol we showed a stabilization time of $\Theta(n)$, a stabilization work of $\Theta(n)$ for $P^{LISTsync}$ and a maintenance work of $\mathcal{O}(1)$ in the STSS model. For the protocols $P^{SMALL-WORLD}$ and $P^{SMALL-WORLDsync}$ we showed a stabilization time of $\mathcal{O}(n + \delta)$ and a stabilization work of $\mathcal{O}(n^2)$, and a maintenance work of $\mathcal{O}(n)$ for the protocol $P^{SMALL-WORLDsync}$ if δ is the converging time of a specific random process. We also presented the protocol P^{CLIQUE} to form a complete graph or a clique and showed a stabilization time of $\Theta(n)$, a stabilization work of $\Theta(n)$, and a maintenance work of $\mathcal{O}(1)$. Although we were able to show upper and lower bounds for P^{LIST} and P^{CLIQUE} it remains open whether the algorithms are optimal. Obviously they are not optimal in the stabilization time but are optimal in the stabilization and maintenance work. An interesting question is how the trade-off between stabilization time and work looks like. It is easy to see that a topology can be formed faster if more messages are sent, but how many messages in one round for one node are optimal to achieve a certain stabilization time is unclear. By now all topologies we presented are based on a one-dimensional order of the identifiers. Open questions are what is possible if there is no order of

the identifiers or if the identifiers are multi-dimensional. Can a 2-dimensional grid be used as an basic topology like we used the sorted list as a basic topology to build a small-world network? Considering the used models and the analysis of topological self-stabilizing protocols it would be interesting to extend the model by properties like monotonicity, i.e. do sub-graphs that are already in a legal state stay in this state during the self-stabilization process or are properties like reachability of a pair of nodes preserved during the stabilization.

CHAPTER 4

Self-Stabilizing Distributed Hash Tables

4.1 Introduction

In this chapter we apply the concept of topological self-stabilization on distributed hash tables. We therefore don't restrict our focus on the topology and the protocol building it any longer, but also consider how data is stored on the nodes forming the topology. In the following we show that our concept of topological self-stabilization can be applied to basic approaches like the Chord network in our solution Re-Chord in section 4.2. By showing that a topological self-stabilizing protocol exists that guarantees to form a Chord-like network out of any weakly connected network, this network is more robust than the original Chord network as it can recover from any failure as long as the network stays weakly connected. Re-Chord maintains all characteristics of the Chord network. That is, a logarithmic routing distance between a pair of nodes, a logarithmic degree and load balancing according to the consistent hashing concept.

For this Re-Chord network we develop a self-stabilizing protocol $P^{Re-Chord}$ that is self-stabilizing in the ATSS model and STSS model with a stabilization time of $\mathcal{O}(n \log n)$, a stabilization work of $\mathcal{O}(n \log^3 n)$, and a maintenance work per round of $\mathcal{O}(\log^2 n)$ for each node. We then proceed by developing a new overlay network and topological self-stabilizing protocol for this network that supports heterogeneity among the nodes in section 4.3. In particular we consider heterogeneity in terms of different capacities of the nodes in our solution CONE-DHT, in which each node has a degree of $\mathcal{O}(\log n)$ w.h.p.. The protocol P^{CONE} for this overlay network works correctly in the ATSS and STSS model. In the STSS model we show a stabilization time of $\mathcal{O}(n)$, a stabilization work of $\mathcal{O}(n^2 + Dn \log n)$ w.h.p. if there are D data items stored in the network, and a maintenance work of $\mathcal{O}(r + \log^2 n)$ w.h.p. if a node stores data items supervised by r reference nodes.

4.1.1 Related Work

Distributed Hash tables are distributed storage systems, i.e. a set of hosts stores a set of data. Distributed Hash Tables (DHTs) were introduced as structured peer-to-peer networks. The most basic and popular concepts are Chord [75], CAN [69], Pastry [70] and tapestry [82]. The common concept of all these DHTs is to map the set of hosts to a virtual address space by a random hash function, such that each host is responsible for a part or region of the address space. To evaluate which host stores which data, the data is also mapped to the address space and stored by the host responsible for the region the data is mapped to. By this the number of data items that need to be moved if a host joins or leaves the system can be limited.

One variant of this idea is the concept of consistent hashing [33] implemented by Chord. In consistent hashing, the data elements are hashed to points in $[0, 1)$ and the hosts are mapped to disjoint intervals in the same interval $[0, 1)$. A host stores all the data elements that are hashed to points in its interval. Consistent hashing was introduced to support the joining and leaving of nodes as w.h.p. at most $\mathcal{O}(K/n)$ keys are moved if a node joins or leaves a network of n nodes and K keys. Furthermore consistent hashing balances the load as each host is responsible for at most $(1 + \epsilon)K/n$ keys in expectation. An alternative strategy is to hash data elements and hosts to pseudo-random bit strings and to store (indexing information about) a data element at the host with the longest prefix match [65]. This concept is implemented by Pastry and Tapestry. The mentioned basic DHTs provide the following features for a set of n hosts:

- logarithmic routing distance, i.e. $O(\log n)$ hops.
- logarithmic degree, i.e. also a routing table of size $O(\log n)$
- (poly-)logarithmic structural changes in case of joining or leaving hosts.
- the maximum load exceeds the average by at most a factor of $O(\log n)$ with high probability.

Other proposed DHT schemes optimize the degree and the routing distance. For example Koorde [32] achieves a routing distance of $O(\log n)$ with a degree of $O(1)$ and a routing distance of $O(\log / \log \log n)$ with a degree of $O(\log n)$.

In a heterogeneous setting, each host (or node) u has its specific capacity $c(u)$ and the goal considered in this paper is to distribute the data among the nodes so that node u stores a fraction of $\frac{c(u)}{\sum_{\forall v} c(v)}$ of the data. The simplest solution would be to reduce the heterogeneous to the homogeneous case by splitting a host of k times the base capacity (e.g., the minimum capacity of a host) into k many virtual hosts. Such a solution is not useful in general because the number of virtual hosts would heavily depend on the capacity distribution, which can create a large management overhead at the hosts. Nevertheless, the concept of virtual hosts has been explored before (e.g., [23, 68, 8]). In [23] the main idea is not to place the virtual hosts belonging to a real host randomly in the identifier space but in a restricted range to achieve a low degree in the overlay network. However, they need an estimation of the network size and a classification of nodes with high, average, and low capacity. A similar approach is presented in [8]. Rao et al. [68] proposed some schemes also based on virtual servers, where the data is moved from heavy nodes to light nodes to balance the load after the data assignment, so and data movement is induced even without joining or leaving nodes. In [74] the authors organize the nodes into clusters, where a super node (i.e., a node with large capacity) is supervising a cluster of nodes with small capacities. Giakkoupis et al. [22] present an approach which focuses on homogeneous networks but also works for heterogeneous one. However, updates can be costly.

Several solutions have been proposed in the literature that can manage heterogeneous storage systems in a centralized way, i.e. they consider data placement strategies for heterogeneous disks that are managed by a single server [71, 54, 51, 16, 81, 11] or assume a central server that handles the mapping of data elements to a set of hosts [72, 12, 13]. We will only focus on the most relevant ones for our approach. In [13] Brinkmann et al. introduced several criteria a placement scheme needs to fulfill, like a faithful distribution, efficient localization, and fast adaptation. They introduce two different data placement strategies named SHARE and SIEVE that fulfill their criteria. To apply their approach, the number of nodes and the overall capacity of the system must be known. In [11] redundancy is added to the SHARE strategy to allow a fair and redundant data distribution, i.e. several copies of a data element are stored such that no two copies are stored on the same host. Another solution to handle redundancy in heterogeneous

systems is proposed in [51], but also here the number of nodes and the overall capacity of the system must be known. The only solution proposed so far where this is not the case is the approach by Schindelhauer and Schomaker [72], which we call *cone hashing*. Their basic idea is to assign a distance function to each host that scales with the capacity of the host. A data element is then assigned to the host of minimum distance with respect to these distance functions. We will extend their construction into a self-stabilizing DHT with low degree and diameter that does not need any global information and that can handle all operations in a stable system efficiently with high probability (w.h.p.)[1].

4.2 Re-Chord: A self-stabilizing Chord overlay network

4.2.1 Introduction

In this section we present a self-stabilizing solution for the Chord network. Chord as one of the pioneering DHT-concepts has been studied thoroughly, and many variants of Chord have been presented that optimize various criteria. Though Chord is known to be very efficient and scalable and it can handle churn quite well, so far no protocol is known yet that guarantees that Chord is self-stabilizing, i.e. the Chord network can be recovered from any initial state in which the network is still weakly connected. We present a extension of the Chord network, called Re-Chord (reactive Chord), that turns out to be locally checkable, and we present a self-stabilizing distributed protocol for it that can recover the Re-Chord network from any initially weakly connected state in $\mathcal{O}(n \log n)$ communication rounds. We also show that our protocol allows a new node to join or to leave an already stable Re-Chord network so that within $\mathcal{O}((\log n)^2)$ communication rounds the Re-Chord network is stable again.

4.2.2 Our contributions

We present a protocol $P^{RE-CHORD}$ solving the overlay problem $RE - CHORD$. The ideas of the protocol $P^{RE-CHORD}$ and the results on stabilization time given in Theorem 4.2.9, stabilization work given in Theorem 4.2.15, maintenance work given in Theorem 4.2.16 and the cases of a single joining or leaving node given in Theorem 4.2.17 are based on results published in [35] and [39] which are joint works with my colleague Andreas Koutsopoulos and our supervisor Christian Scheideler:

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler, Re-Chord: A Self-stabilizing Chord Overlay Network, 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), San Jose, California, USA, June 2011.

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler, Re-Chord: A Self-stabilizing Chord Overlay Network. Theory of Computing Systems. 55(3): 591-612 (2014)

The results for the correctness in the ATSS model in Theorem 4.2.2 are presented for the first time.

In this section we show that the protocol $P^{RE-CHORD}$ solves the $RE-CHORD$ overlay problem in the ATSS and STSS model with a stabilization time of $\mathcal{O}(n \log n)$ and a stabilization work of $\mathcal{O}(n \log^3 n)$. We further show a maintenance work per round of $\mathcal{O}(\log^2 n)$ for each node once a legal state has been reached. We also consider topology updates caused by a single joining or leaving node and show that the network recovers in $\mathcal{O}(\log^3 n)$ rounds with at most $\mathcal{O}(\log^2 n)$ messages for each node.

[1] I.e., a probability of $1 - n^{-c}$ for any constant $c > 0$

4.2.3 Chord

Chord is basically a combination of a hypercubic overlay network with consistent hashing [33]. Consistent hashing assigns keys to nodes by first hashing the nodes and then hashing the keys to the same space by a hash function h that assigns nodes to the space uniformly at random (SHA-1 in Chord). A key k is assigned to the node u, such that $h(u)$ is equal to or follows $h(k)$. In Chord h maps to a circle of numbers from 0 to $2^m - 1$ so u is the next node following $h(k)$ in clockwise order.

The Chord overlay network is defined in the following way [75]: Let $successor(h(u))$ denote the node following $h(u)$ in clockwise order on the circle of numbers from 0 to $2^m - 1$. Then each node u stores a pointer to $successor(h(u) + 2^{i-1} \text{mod} 2^m)$ for $1 \leq i \leq m$. Such a node is called the ith finger of u. The first finger $successor(u)$ obviously is the successor of u on the circle. Additionally to the fingers each node also stores its predecessor on the circle. Chord then provides a routing length of $\mathcal{O}(\log n)$ w.h.p., i.e. in $\mathcal{O}(\log n)$ hops $successor(h(k))$, the node responsible for k is reached for a key k. Furthermore the degree of each node is bounded by $m = \mathcal{O}(\log n)$. If a node joins or leaves a Chord network it takes $\mathcal{O}(\log^2 n)$ messages to re-establish the finger tables and routing invariants. An illustration of the $m - 1$ fingers of a node u is given in Figure 4.1.

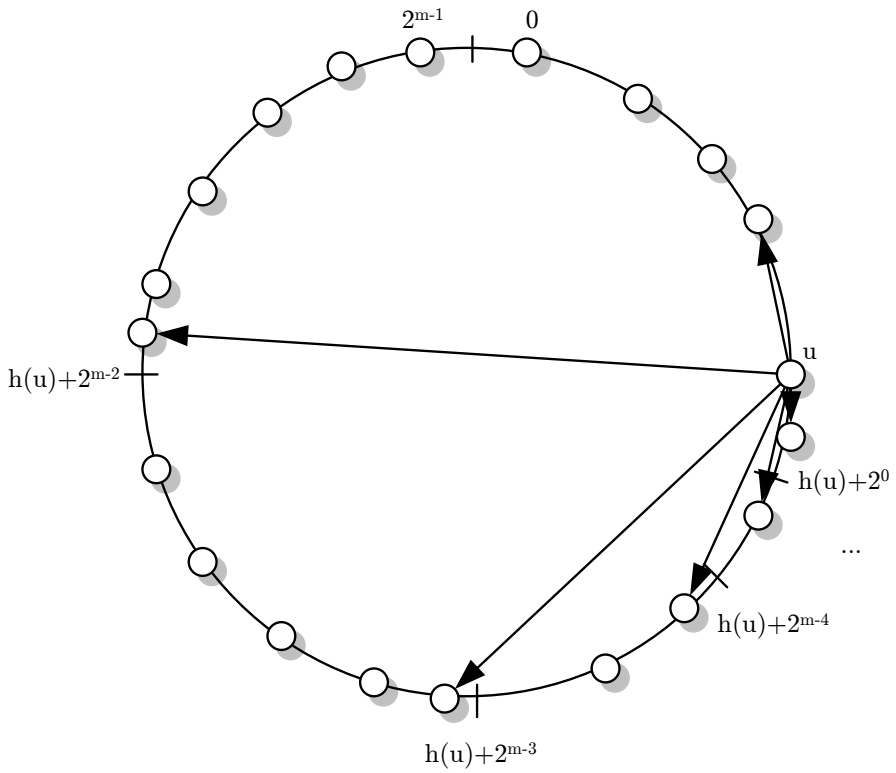


Figure 4.1: An example of the m finger in a Chord network

The authors also point out that the Chord network can be made more robust if each node stores not only its successor but the next $r = \mathcal{O}(\log n)$ nodes succeeding it. However the disadvantage of Chord is that there are states that Chord cannot recover from, e.g. if the nodes are partitioned in two disjoint circles that are only connected by fingers (see Figure 4.2). Such a state cannot be locally detected and repaired

as for every node locally the fingers and successor and predecessors are correct. Our solution is able to recover from any weakly connected state even if connectivity is only given by messages containing identifiers of nodes.

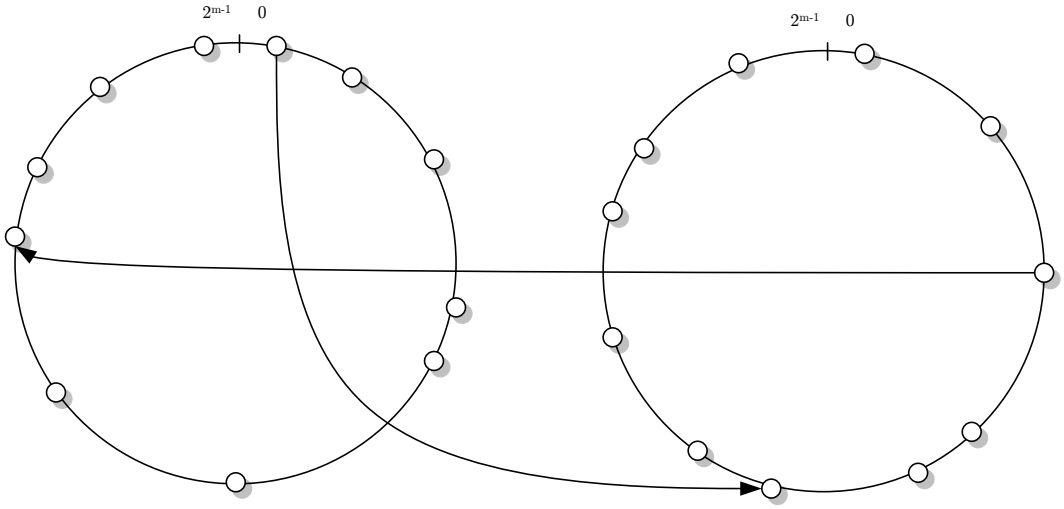


Figure 4.2: A state Chord cannot recover from

Several variants of Chord have already been studied since the presentation of the Chord network. In [46] a variant called EPI Chord is presented that allows the system to do parallel searches for the best route to the node storing the data for a certain search key. This does not improve the asymptotical worst-case cost of $\mathcal{O}(\log n)$ messages of Chord but it can achieve $\mathcal{O}(1)$ hop look-up performance under look-up intensive workloads due to caching. In [52] another modification of Chord is presented. In this approach Chord is extended by symmetric fingers, hence one can search in both directions of the circle. A similar idea is given in [31] and [78], where links to the predecessors are stored instead of only links to the successors of a node. In [78] also the physical distance is taken into account to estimate the shortest route. All these variants only care about the look-up cost, but present no self-stabilizing process to maintain the Chord structure. In [55] an algorithm is presented to build a Chord network from scratch in $\mathcal{O}(\log n)$ rounds, but still this algorithm is not self-stabilizing.

4.2.4 The Re-Chord Network

Our Re-Chord network is able to emulate the Chord network while maintaining all its properties and can be maintained by a topological self-stabilizing protocol, $P^{RE-CHORD}$. It can recover from any failure as long as the network stays weakly connected. The idea is to reduce the problem of building a Chord-like network to the sorted list problem presented in Section 3.2. To be more general then the Chord network we assume $h : ID \mapsto [0, 1]$ to be a randomized hash function and let it map to the $[0, 1]$ interval instead of the numbers from 0 to $2^m - 1$. Then the number of nodes is not longer restricted by 2^m. We assume that h is chosen, such that no two nodes get the same position and that h is known to every node, i.e. by knowing the identifier of one node each node can determine its position. Furthermore h has to be a random hash function, i.e. the position of each node is chosen uniformly at random. Then a global order on the nodes can be defined by their positions $h(v_1) < h(v_2) \cdots < h(v_n)$. To reduce the Re-Chord

problem to the sorted list problem each node simulates a number of *virtual* nodes instead of edges to the finger nodes. Each virtual node than has an edge to its succeeding and preceding *real* node.

Definition 4.2.1 *We call a node $u \in V$ a* real *node. Each real node simulates a set of virtual nodes u^i with $h(u^i) = h(u) + \frac{1}{2^i} \mod 1$ for $0 \leq i \leq m(u)$. Then $u^0 = u$. We denote the set of all (virtual) nodes by V^+. We call all nodes u^i belonging to the same real node u* siblings*. We choose $m(u) = \min\left\{i \in \mathbb{N} : \nexists v \in V \wedge v \in \left(u, u + \frac{1}{2^i}\right]\right\}$.*

Note that we use the notation of u^i to denote a virtual node and $x_0, \cdots, x_i$ to denote the ith node in some sequence of nodes. To define the Re-Chord network we first define some relations between the nodes in the network:

Definition 4.2.2 *We define*

- $successor_r(u^i) = \textit{argmin}\left\{h(v) : v \in V : \wedge\, h(v) > h(u^i)\right\}$
- $successor_v(u^i) = \textit{argmin}\left\{h(v^j) : v^j \in V^+ : \wedge\, h(v^j) > h(u^i)\right\}$
- $predecessor_r(u^i) = \textit{argmax}\left\{h(v) : v \in V : \wedge\, h(v) < h(u^i)\right\}$
- $predecessor_v(u^i) = \textit{argmax}\left\{h(v^j) : v^j \in V^+ : \wedge\, h(v^j) < h(u^i)\right\}$
- $cycle_r(u^i) = \textit{argmax}\left\{h(v) : v \in V\right\}$ *if* $predecessor_r(u^i) = nil$.
- $cycle_r(u^i) = \textit{argmin}\left\{h(v) : v \in V\right\}$ *if* $successor_r(u^i) = nil$).
- $cycle_v(u^i) = \textit{argmax}\left\{h(v) : v \in V^+\right\}$ *if* $predecessor_v(u^i) = nil$.
- $cycle_v(u^i) = \textit{argmin}\left\{h(v) : v \in V^+\right\}$ *if* $successor_v(u^i) = nil$).

Then a graph $G = (V, E_r)$ is a Re-Chord network, if $V^+ = \bigcup_{u \in V}\left\{u^0, \ldots \cdots u^{m(u)}\right\}$ and $E_r = \left\{(u, v) : v \in \bigcup_{0 \leq i \leq m(u)}\left\{predecessor_r(u^i), successor_r(u^i), cycle_r(u^i)\right\}\right\}$.

We show that we can emulate a Chord network by the above defined Re-Chord network, as Chord is a sub-graph of a Re-Chord network.

Theorem 4.2.1 *Chord is a sub-graph of a Re-Chord network. In the Re-Chord network each real node simulates $\mathcal{O}(\log n)$ virtual nodes w.h.p. probability and $|V^+| = \mathcal{O}(n \log n)$ w.h.p. and $|E_{Re-Chord}| \leq 4|E_{Chord}|$.*

Proof. Let (u, v) be an edge in the original Chord network, i.e. $(u, v) \in E_{Chord}$. If $v = successor(h(u))$ or $v = predecessor(h(u))$ in the Chord network, then $v = successor_r(u)$ or $v = predecessor_r(u)$ in the Re-Chord network. If v is a finger of u in the Chord network then it holds that $v = successor_r(u^i)$ in the Re-Chord network for some virtual node u^i. In fact the mth finger in the Chord network corresponds to $v = successor_r(u^0)$ the $m-1$th finger to $v = successor_r(u^1)$ and the $m - i$th finger to $v = successor_r(u^{1+i})$ (see Figure 4.3).

We can show that each node simulates $\mathcal{O}(\log n)$ virtual nodes w.h.p. corresponding to the $\mathcal{O}(\log n)$ fingers in the Chord network even without knowing n. Remind that in the Chord network m and therefore the maximal number of nodes is fixed. If the positions of all nodes are distributed uniformly

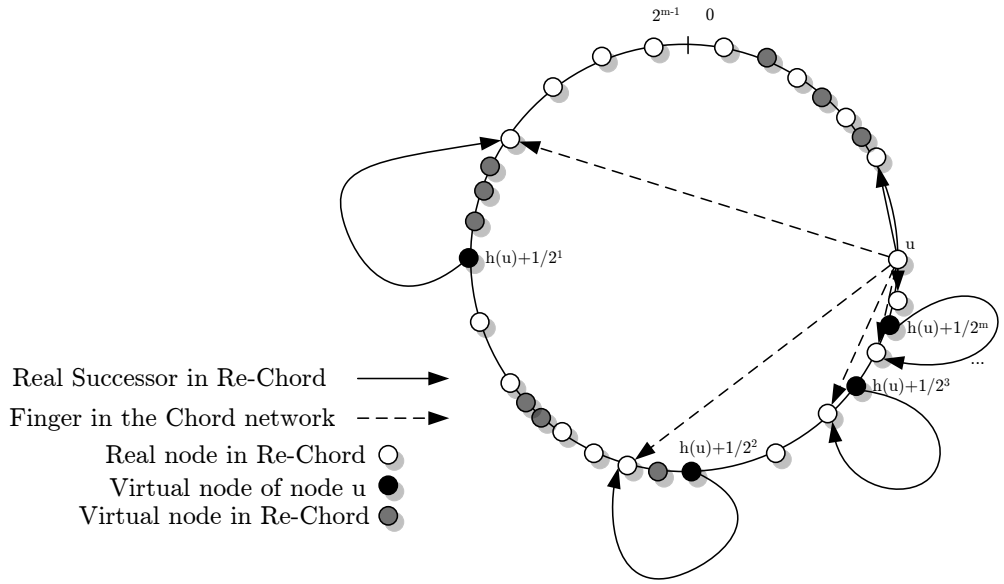


Figure 4.3: Each finger in Chord is simulated in Re-Chord by a real successor of a virtual node

at random then w.h.p. the distance between $h(u^0)$ and $h(successor_r(u^0))$ is at least $\frac{1}{n^2}$. Then the distance between $h(u^0)$ and $h(u^m(u))$ is at least $\frac{1}{n}$ w.h.p. and therefore $m(u) = \mathcal{O}(\log n)$. As each real node simulate $\mathcal{O}(\log n)$ virtual nodes in expectation there are $\mathcal{O}(\log n)$ virtual nodes between u^0 and $successor_r(u^0)$. By applying a Chernoff bound one can also show that this holds w.h.p.. Then follows that $V^+ = \mathcal{O}(n \log n)$ w.h.p..

Thus all edges in Chord E_{Chord} can also be found in Re-Chord $E_{Re-Chord}$, which also has some additional edges as we also store predecessors for every virtual node, and $E_{Chord} \subset E_{Re-Chord}$. In fact $|E_{Re-Chord}| \leq 4|E_{Chord}|$ as we store for each virtual node a real and a virtual predecessor and successor. □

Then also for Re-Chord it holds that the degree of nodes is bounded by $\mathcal{O}(\log n)$ and the routing length is $\mathcal{O}(\log n)$.

4.2.5 Formal Definition

We now define the internal variables of each node used in our protocol to solve the overlay problem $RE - CHORD$. Each real node u stores the following variables contributing to the explicit edge set:

- $u.predecessor_r$: The identifier of u's left neighbor of real nodes($h(u.predecessor_r) < h(u)$ with $u.predecessor_r \in V$) or nil if there is no left real neighbor.
- $u.predecessor_v$: The identifier of u's left virtual neighbor ($h(u.predecessor) < h(u)$ with $u.predecessor_r \in V^+$) or nil if there is no left neighbor.
- $u.successor_r$: The identifier of u's right real neighbor ($h(u) < h(u.successor)$ with $u.successor_r \in V$) or nil if there is no right neighbor.

- $u.successor_v$: The identifier of u's right virtual neighbor ($h(u) < h(u.successor)$ with $u.successor_v \in V$) or nil if there is no right neighbor.
- $u.cycle_r$: The identifier of the real node that u's cycle edge points to. Note that this identifier is only set if $u.successor_r = nil$ or $u.predecessor_r = nil$.
- $u.cycle_v$: The identifier of the virtual node that u's cycle edge points to. Note that this identifier is only set if $u.successor_v = nil$ or $u.predecessor_v = nil$.
- $u.U^+$: the set of virtual nodes $\{u^1, \cdots u^{u.m}\}$.
- $u.N = \{u.predecessor_r, u.predecessor_v, u.successor_r, u.successor_v, u.cycle_r, u.cycle_v\} \cup u.U^+$: The neighborhood of one node.

Additionally each real node stores the following variables not contributing to the set of explicit edges.

- *u.m*: the limit of virtual nodes m such that $m = \text{argmin}_{i \in \mathbb{N}} \{h(u^i) : u.successor_r \notin [u, u^i]\}$ (resp. $m = \text{argmin}_{i \in \mathbb{N}} \{h(u^i) : u.cycle_r \notin [u, u^i]\}$).
- *u.Ch*: The channel for incoming messages.
- $u.\tau$: a boolean variable that is periodically true.

Each virtual node u^i for some node u and $i > 0$ stores the same variables as a real node except the set of virtual nodes and the limit of virtual nodes m, although we assume that each virtual node is aware of all its siblings. This is reasonable as all virtual nodes u^i are simulated by the same real node u.

The content of a message is given by the ids of some nodes and a specification of the message type. So different type of messages can trigger different actions. Thus a message m is of the following form $m = (type, ids)$.

- *forward*: The standard message type to forward a node to another node.
- *cycle-neighbor-real*: This type of message is used to establish a cycle edge for the neighbors of a node if a node is missing its left or right real neighbor.
- *cycle-virtual*: This type of message is used to establish a cycle edge if a node is missing its left or right virtual neighbor.
- *respond-cycle-virtual*: This message is sent to respond to an incoming cycle edge and to inform the start-point of the cycle edge about possible network changes.
- *probing*: This type of message is used to propagate a probing message.

In the following we define when an assignment of the variables is valid and how the sets of initial topologies IT and the goal topologies $RE - CHORD$ look like.

Definition 4.2.3 *An assignment of the variables of a node is valid if* $h(u^i.predecessor_v) < h(u^i)$ *and* $h(u^i.predecessor_r) \leq h(u^i.predecessor_v)$ *(or* $u^i.predecessor_r = nil$ *and* $u^i.predecessor_v = nil$*) and* $h(u^i.successor_r) \geq h(u^i.successor_v) > h(u^i)$ *(or* $u^i.successor_r = nil$ *and* $u^i.successor_v = nil$*). Furthermore* $u^i.cycle_r \neq nil$ *only if* $u^i.successor_r = nil$ *or* $u^I.predecessor_r = nil$ *and*

$u^i.cycle_v \neq nil$ only if $u^i.successor_v = nil$ or $u^i.predecessor_v = nil$. Also m is valid if $m = argmin_{i\in\mathbb{N}} \{h(u^i) : u.successor_r \notin [u, u^i]\}$ (resp. $m = argmin_{i\in\mathbb{N}} \{h(u^i) : u.cycle_r \notin [u, u^i]\}$). Note that an invalid assignment can be locally repaired immediately. Thus we assume in the following w.o.l.g. that initially the assignment is valid for every node.

Definition 4.2.4 *Let E_e and E_i be defined as described in Chapter 2 according to the definition of internal variables. Then the set of initial topologies is given by:*

$$IT = \{G = (V^+, E = E_e \cup E_i) : G \text{ is weakly connected}\}$$

Definition 4.2.5 *Let $E_r = \{(u, v) : v \in \bigcup_{u^i\in u-U^+} \{u^i.predecessor_r, u^i.successor_r, u^i.cycle_r\}\}$. For a graph $G = (V^+, E)$ we denote by $G_r(G) = (V, E_r)$ the graph only formed by real nodes. Then the set of target topologies is given by:*

$$RE - CHORD = \{G = (V^+, E) : G_r(G) \text{ is a Re-Chord network}\}$$

4.2.6 Protocol $P^{RE-CHORD}$

In this section we give a description of our algorithm. The algorithm is a protocol that each node executes based on its own node and channel state. The protocol contains periodic actions that are executed if the timer predicate τ is true and actions that are executed if the node receives a message m.

As mentioned above we assume that all internal variables are valid at every time, that means in particular that each real node computes the correct value of $u.m$ and simulates the corresponding virtual nodes $u^0, \cdots u^{u.m} \in u.U^+$.

The main idea of the protocol $P^{RE-CHORD}$ is that each node whether real or virtual executes similar actions as presented in P^{LIST} to form a sorted list. The actions in $P^{RE-CHORD}$ are more complex as we have to distinguish between received identifiers of real and virtual nodes. Like in $P^{SMALL-WORLD}$ we also extend the protocol to build a cyclic list instead of a simple sorted list. Again we distinguish between real and virtual nodes when closing the cycle. As we use the virtual siblings of a node as shortcuts in the forwarding process while building a sorted list (see Figure 4.4) we have to ensure that all $u^i \in u.U^+$ belonging to a real node u form one connected component. To ensure this connectivity we reuse the probing technique presented in Section 3.3. We also use the probing to ensure connectivity between a node and the node its cycle edge points to.

We present an implementation in Pseudo code of $P^{RE-CHORD}$ in Algorithms 4.2.1- 4.2.8.

4.2.7 Analysis in the ATSS model

We show the correctness in the ATSS model of our approach by proving that it stabilizes to a small-world network. We show this by dividing the self-stabilization process into different phases and determine the correctness of each phase. In particular we will show that if each node executes $P^{RE-CHORD}$ started in an weakly connected graph, eventually a graph is reached, such that the sorted list is sub graph. We then show that if started in a graph that contains the sorted list as a sub graph $P^{RE-CHORD}$ leads to a graph containing the cyclic list as a sub graph. We proceed by showing that started in a graph that contains the cyclic list a a sub graph eventually a Re-Chord network is reached. Thus we can show the main theorem:

Theorem 4.2.2 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORD}$ then eventually the graph converges to a graph $G' \in RE - CHORD$ (Convergence).*

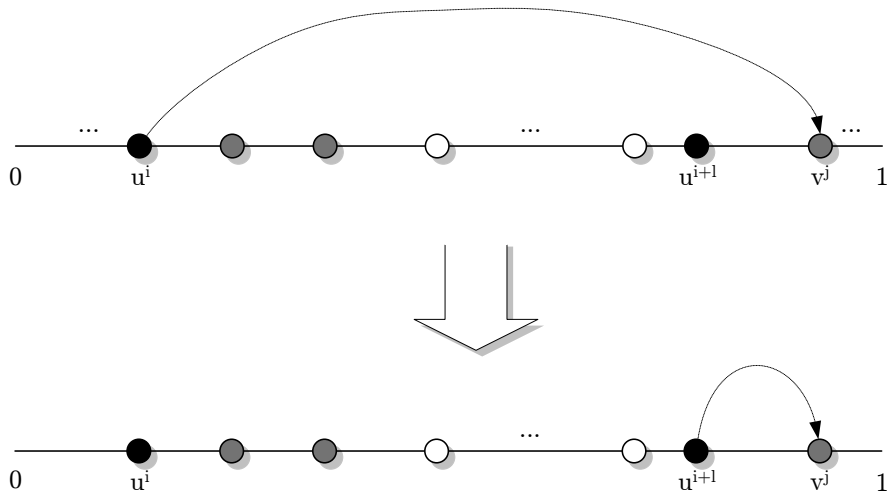


Figure 4.4: u^i forwards v^j to its virtual sibling u^{i+l}

Algorithm 4.2.1 $P^{RE-CHORD}$ FOR AN NODE u^i

$u^i.N = \{u^i.successor_v, u^i.successor_r, u^i.cycle_v, u^i.cycle_r, u^0, \cdots u^{u.m}\}$
message $m = (type, v) \in u^i.Ch \rightarrow$
if type=forward **then**
    forward(v)
else if type=probing **then**
    probing(v)
else if type=cycle-virtual **then**
    cycle-virtual(v)
else if type=respond-cycle-virtual **then**
    respond-cycle-virtual(v)
else if type=cycle-neighbor-real **then**
    cycle-neighbor-real(v)
$\tau \rightarrow$
introduction()
probing()

Algorithm 4.2.2 $P^{RE-CHORD}$-FORWARD(v)

$F = \emptyset$ ▷ set of nodes to be forwarded
if $h(v) > h(u^i)$ **then**
  if v is real **then**
    if $u^i.successor_r \neq nil$ **then**
      if $h(v) < h(u^i.successor_r)$ **then**
        $F = \cup \{u^i.successor_r\}$
        $u^i.successor_r = v$
      else
        $F = F \cup \{v\}$
    else
      $u^i.successor_r = v$
      if $u^i.cycle_r \neq nil$ **then**
        $F = F \cup \{u^i.cycle_r\}$
        $u^i.cycle_r = nil$
  if $u^i.successor_v \neq nil$ **then**
    if $h(v) < h(u^i.successor_v)$ **then**
      $F = \cup \{u^i.successor_v\}$
      $u^i.successor_v = v$
    else
      $F = F \cup \{v\}$
  else
    $u^i.successor_v = v$
    if $u^i.cycle_v \neq nil$ **then**
      $F = F \cup \{u^i.cycle_v\}$
      $u^i.cycle_v = nil$
  for all $w \in F$ **do**
    if $h(w) > h(u^i)$ **then**
      $w' = \text{argmax} \{h(w') : h(w') < h(w) \wedge w' \in u^i.N\}$
      send message $m = (forward, w)$ to w'
    else
      $w' = \text{argmin} \{h(w') : h(w') > h(w) \wedge w' \in u^i.N\}$
      send message $m = (forward, w)$ to w'

else if $h(v) < h(u^i)$ **then**
  if v is real **then**
    if $u^i.predecessor_r \neq nil$ **then**
      if $h(v) < h(u^i.predecessor_r)$ **then**
        $F = \cup \{u^i.predecessor_r\}$
        $u^i.predecessor_r = v$
      else
        $F = F \cup \{v\}$
    else
      $u^i.predecessor_r = v$
      if $u^i.cycle_r \neq nil$ **then**
        $F = F \cup \{u^i.cycle_r\}$
        $u^i.cycle_r = nil$
  if $u^i.predecessor_v \neq nil$ **then**
    if $h(v) < h(u^i.predecessor_v)$ **then**
      $F = F \cup \{u^i.predecessor_v\}$
      $u^i.predecessor_v = v$
    else
      $F = F \cup \{v\}$
  else
    $u^i.predecessor_v = v$
    if $u^i.cycle_v \neq nil$ **then**
      $F = F \cup \{u^i.cycle_v\}$
      $u^i.cycle_v = nil$
  for all $w \in F$ **do**
    if $h(w) > h(u^i)$ **then**
      $w' = \text{argmax} \{h(w') : h(w') < h(w) \wedge w' \in u^i.N\}$
      send message $m = (forward, w)$ to w'
    else
      $w' = \text{argmin} \{h(w') : h(w') > h(w) \wedge w' \in u^i.N\}$
      send message $m = (forward, w)$ to w'

Algorithm 4.2.3 $P^{RE-CHORD}$-PROBING(v)

if $h(v) > h(u^i)$ **then**
  if $\exists w : h(w) < h(v) \wedge w \in \{u^i.successor_v, u^i.cycle_v, u.U^+\}$ **then**
    $w' = \text{argmax}\{h(w) : h(w) < h(v) \wedge w \in \{u^i.successor_v, u^i.cycle_v\} \cup u.U^+\}$
    send message $m = (probing, v)$ to w'
  else
    forward(v)
else
  if $\exists w : h(w) > h(v) \wedge w \in \{u^i.predecessor_v, u^i.cycle_v, u^0, \cdots u^{u.m}\}$ **then**
    $w' = \text{argmin}\{h(w) : h(w) > h(v) \wedge w \in \{u^i.predecessor_v, u^i.cycle_v\} \cup u.U^+\}$
    send message $m = (probing, v)$ to w'
  else
    forward(v)

Algorithm 4.2.4 $P^{RE-CHORD}$-CYCLE-VIRTUAL(v)

if $h(v) < h(u^i)$ **then**
  if $\exists w : h(w) < h(v) \wedge w \in u^i.N$ **then**
    $w = \text{argmax}\{h(w') : h(w') < h(v) \wedge w' \in u^i.N\}$
    send message $m = (forward, w)$ to v
  else if $\exists w : h(w) > h(v) \wedge w \in u^i.N$ **then**
    $w = \text{argmax}\{h(w') : h(w') > h(v) \wedge w' \in u^i.N\}$
    send message $m = (respond - cycle - virtual, w)$ to v
  else
    send message $m = (respond - cycle - virtual, u^i)$ to v
else
  if $\exists w : h(w) > h(v) \wedge w \in u^i.N$ **then**
    $w = \text{argmin}\{h(w') : h(w') > h(v) \wedge w' \in u^i.N\}$
    send message $m = (forward, w)$ to v
  else if $\exists w : h(w) < h(v) \wedge w \in u^i.N$ **then**
    $w = \text{argmin}\{h(w') : h(w') > h(v) \wedge w' \in u^i.N\}$
    send message $m = (respond - cycle - virtual, w)$ to v
  else
    send message $m = (respond - cycle - virtual, u^i)$ to v

Algorithm 4.2.5 $P^{RE-CHORD}$-RESPOND-CYCLE-VIRTUAL(v)

if $u^i.predecessor_v = nil \wedge (h(v) > h(u^i.cycle_v) > h(u^i) \vee (u^i.cycle_v = nil \wedge h(v) > h(u^i)))$ **then**
  if $u^i.cycle_v \neq nil$ **then**
    $w = u^i.cycle_v$
  $u^i.cycle_v = v$
  $forward(w)$
else if $u^i.successor_v = nil \wedge (h(v) < h(u^i.cycle_v) < h(u^i) \vee (u^i.cycle_v = nil \wedge h(v) < h(u^i)))$ **then**
  if $u^i.cycle_v \neq nil$ **then**
    $w = u^i.cycle_v$
  $u^i.cycle_v = v$
  $forward(w)$
elseforward(v)

Algorithm 4.2.6 $P^{RE-CHORD}$-CYCLE-NEIGHBOR-REAL(v)

if v is real **then**
  if $u^i.predecessor_r = nil \wedge (h(v) > h(u^i.cycle_r) > h(u^i) \vee (u^i.cycle_r = nil \wedge h(v) > h(u^i)))$ **then**
    if $u^i.cycle_r \neq nil$ **then**
      $w = u^i.cycle_r$
    $u^i.cycle_r = v$
    forward(w)
  else if $u^i.successor_r = nil \wedge (h(v) < h(u^i.cycle_r) < h(u^i) \vee (u^i.cycle_r = nil \wedge h(v) < h(u^i)))$ **then**
    if $u^i.cycle_r \neq nil$ **then**
      $w = u^i.cycle_r$
    $u^i.cycle_r = v$
    forward(w)
  elseforward(v)
else
  forward(v)

Algorithm 4.2.7 $P^{RE-CHORD}$-INTRODUCTION()

if $u^i.predecessor_v \neq nil$ **then** ▷ u^i introduces itself to $u^i.predecessor_v$
  send message $m = (forward, u^i)$ to $u^i.predecessor_v$
else ▷ u^i introduces itself to $u^i.cycle_v$
  send message $m = (cycle - virtual, u^i)$ to $u^i.cycle_v$
if $h(u^i.successor_v) \neq nil$ **then** ▷ u^i introduces itself to $u^i.successor_v$
  send message $m = (forward, u^i)$ to $u^i.successor_v$
else ▷ u^i introduces itself to $u^i.cycle_v$
  send message $m = (cycle - virtual, u^i)$ to $u^i.cycle_v$
if $h(u^i.predecessor_r) \neq nil$ **then** ▷ real predecessor is forwarded to all virtual nodes in between
  send message $m = (forward, u^i.predecessor_r)$ to $u^i.predecessor_v$
  if u^i is not real **then**
    if $u^i.successor_v \neq nil$ **then**
      send message $m = (forward, u^i.predecessor_r)$ to $u^i.successor_v$
    else
      send message $m = (cycle - neighbor - real, u^i.predecessor_r)$ to $u^i.cycle_v$
else ▷ real predecessor ($u^i.cycle_r$) is forwarded to all virtual nodes in between
  send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.predecessor_v$
  if u^i is not real **then**
    send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.successor_v$
if $h(u^i.successor_r) \neq nil$ **then** ▷ real successor is forwarded to all virtual nodes in between
  send message $m = (forward, u^i.successor_r)$ to $u^i.successor_v$
  if u^i is not real **then**
    if $u^i.predecessor_v \neq nil$ **then**
      send message $m = (forward, u^i.successor_r)$ to $u^i.predecessor_v$
    else
      send message $m = (cycle - neighbor - real, u^i.successor_r)$ to $u^i.cycle_v$
else ▷ real successor given by $u^i.cycle_r$ is forwarded to all virtual nodes in between
  send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.successor_v$
  if u^i is not real **then**
    send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.predecessor_v$

Algorithm 4.2.8 $P^{RE-CHORD}$-PROBING()

if $\exists u^j : j > i$ **then** ▷ Probing to next virtual sibling u^j
  if $h(u^j) < h(u^i.predecessor_v)$ **then**
    send message $m = (probing, u^j)$ to $u^i.predecessor_v$
  else if $h(u^j) > h(u^i.successor_v)$ **then**
    send message $m = (probing, u^j)$ to $u^i.successor_v$
if $u^i.cycle_v \neq nil \wedge h(u^i.cycle_v) < h(u^i.predecessor_v)$ **then** ▷ Probing to $u^i.cycle_v$
  send message $m = (probing, u^i.cycle_v)$ to $u^i.predecessor_v$
else if $u^i.cycle_v \neq nil \wedge h(u^i.cycle_v) > h(u^i.successor_v)$ **then**
  send message $m = (probing, u^i.cycle_v)$ to $u^i.successor_v$
if $u^i.cycle_r \neq nil \wedge h(u^i.cycle_r) < h(u^i.predecessor_v)$ **then** ▷ Probing to $u^i.cycle_r$
  send message $m = (probing, u^i.cycle_r)$ to $u^i.predecessor_v$
else if $u^i.cycle_r \neq nil \wedge h(u^i.cycle_r) > h(u^i.successor_v)$ **then**
  send message $m = (probing, u^i.cycle_r)$ to $u^i.successor_v$

If in an initial graph $G \in RE - CHORD$ every nodes executes the protocol $P^{RE-CHORD}$ then each possible computation leads to a graph $G' \in RE - CHORD$ (Closure).

To prove this theorem we have to introduce some formal definition of the intermediate graphs and some subset of the edges.

Definition 4.2.6 *We define the following subsets of edges.*

- $E_{e-list} = \{(u,v) \in E_e : v = u.predecessor_v \vee v = u.successor_v\} \subseteq E_e$ *is the subset of explicit edges that will form the sorted list.*
- $E_{i-list} = \{(u,v) \in E_i : \exists m = (forward, v) \in u.Ch\}$ *is the subset of implicit edges that are needed to form the sorted list.*
- $E_{list} = E_{e-list} \cup E_{i-list}$ *is the set of edges that take part in the process forming the sorted list.*
- $E_{e-cycle} = \{(u,v) \in E_e : v = u.cycle_v\} \cup E_{e-list} \subseteq E_e$ *is the subset of explicit edges that will form the cyclic list.*
- $E_{i-cycle} = E_{i-list} \cup \{(u,v) \in E_i : \exists m = (type, v) \in u.Ch\}$ *with* $type = cycle - virtual$ *or* $type = respond - cycle$. $E_{i-cycle}$ *is the subset of implicit edges that are needed to form the cyclic list.*
- $E_{cycle} = E_{e-cycle} \cup E_{i-cycle}$ *is the set of edges that take part in the process of forming the cyclic list.*

Note that the above edge sets are defined on V^+ instead of V, i.e. we want to build a sorted list resp. a cyclic list considering all virtual nodes instead of real nodes only.

Definition 4.2.7 *A graph $G = (V, E)$ is a sorted list, if $V = v_0, \cdots v_{n-1}$ with $h(v_i) < h(v_{i+1}) \forall i \in \{0, \cdots n-1\}$ and $E = \{(v_i, v_j) :, j = i-1 \vee j = i+1 \forall 0 < j < n-1\}$.*

Definition 4.2.8 *A graph $G = (V, E)$ is a cyclic list, if $V = v_0, \cdots v_{n-1}$ with $h(v_i) < h(v_{i+1}) \forall i \in \{0, \cdots n-1\}$ and $E = \{(v_i, v_{i+1 \mod n}), (v_i, v_{i-1 \mod n})\}$.*

Convergence

We start by showing that the first part of theorem 4.2.2 holds and $P^{RE-CHORD}$ fulfills the convergence property.

Theorem 4.2.3 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORD}$ then eventually the graph converges to a graph G' such that $G_r(G') \in RE - CHORD$ (Convergence).*

To show this we prove a set of intermediate theorems first, from which the convergence follows. Our first theorem claims that the computation reaches a state, such that the sorted list is a sub graph of the graph of explicit edges.

Theorem 4.2.4 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORD}$ then eventually the computation reaches a state s_t such that G^t_{e-list} is a sorted list.*

Lemma 4.2.1 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORD}$ then eventually the computation reaches a state s_t such that G^t_{list} is weakly connected and stays connected in every state $s_{t'}$ $t' > t$.*

Proof. We start the proof by showing that if G^t is weakly connected also G^{t+1} stays weakly connected. We therefore assume that an edge $(u, v) \in E^t$ existing at time t does not exist at time $t + 1$ and show that u and v stay connected. We consider any possible explicit and implicit edge. The basic idea is to show that if an identifier v of a node is not stored by a node u then it is forwarded to another node w that is connected to u. Thus no identifier can get lost.

If $(u^i, v^j) \in E^t_e$ but $(u^i, v^j) \notin E^{t+1}$ and

- $v^j = u^i.successor_v$ at time t but not at time $t + 1$, then v^j is replaced by a new virtual successor $w^k = u^i.successor_v$. Then u^i forwards v^j to a node $w'^l \in u^i.N$ and $(u^i, w'^l) \in E^{t+1}_e$ and $(w'^l, v^j) \in E^{t+1}_i$. Thus u^i and v^j stay connected.

- $v^0 = u.successor_r$ at time t but not at time $t + 1$, then v^0 is replaced by a new real successor $w^0 = u^i.successor_r$. Then u^i forwards v^0 to a node $w'^l \in u^i.N$ and $(u^i, w'^l) \in E^{t+1}_e$ and $(w'_l, v^0) \in E^{t+1}_i$. Thus u^i and v^0 stay connected.

- $v^j = u^i.predecessor_v$ at time t but not at time $t+1$, then v^j is replaced by a new virtual predecessor $w^k = u^i.predecessor_v$. Then u^i forwards v^j to a node $w'^l \in u^i.N$ and $(u^i, w'^l) \in E^{t+1}_e$ and $(w'^l, v^j) \in E^{t+1}_i$. Thus u^i and v^j stay connected.

- $v^0 = u^i.predecessor_r$ at time t but not at time $t + 1$, then v^0 is replaced by a new real predecessor $w^0 = u^i.successor_v$. Then u^i forwards v^0 to a node $w'_l \in u^i.N$ and $(u^i, w'_l) \in E^{t+1}_e$ and $(w'_l, v^0) \in E^{t+1}_i$. Thus u^i and v^0 stay connected.

- $v^j = u^i.cycle_v$ at time t but not at time $t + 1$, then v^j is replaced by a new virtual cycle node $w^k = u^i.cycle_v$ or $u^i.cycle_v = nil$ as u^i is no longer missing a virtual predecessor or successor. Then u^i forwards v^j to a node $w'^l \in u^i.N$ and $(u^i, w'^l) \in E^{t+1}_e$ and $(w'^l, v^j) \in E^{t+1}_i$. Thus u^i and v^j stay connected.

- $v^0 = u^i.cycle_r$ at time t but not at time $t+1$, then v^0 is replaced by a new real cycle node $w^0 = u^i.cycle_r$ or $u^i.cycle_r = nil$ as u^i is no longer missing a real predecessor or successor. Then u^i forwards v^0 to a node $w'^l \in u^i.N$ and $(u^i, w'^l) \in E_e^{t+1}$ and $(w'^l, v^0) \in E_i^{t+1}$. Thus u^i and v^0 stay connected.

If $(u^i, v^j) \in E_i^t$ but $(u^i, v^j) \notin E^{t+1}$, then u^i does not store v^j in one of its internal variables. Then the following observations hold. If

- v^j is stored in a forward message, then u^i receives this message and forwards v^j to a node $w^k \in u^i.N$. Then $(u^i, w^k) \in E_e^{t+1}$ and $(w^k, v^j) \in E_i^{t+1}$ and u^i and v^j stay connected.
- v^j is stored in a cycle-virtual message, then u^i responds by a respond-cycle-virtual message containing a node $w^k \in u^i.N$ to v^j. Then $(u^i, w^k) \in E_e^{t+1}$ and $(v^j, w^k) \in E_i^{t+1}$ and u^i and v^j stay connected.
- v^j is stored in a respond-cycle-virtual message, then as u^i does not store v^j, v^j is forwarded in a forward message to a node $w^k \in u^i.N$. Then $(u^i, w^k) \in E_e^{t+1}$ and $(v^j, w^k) \in E_i^{t+1}$ and u^i and v^j stay connected.
- v^j is stored in a cycle-neighbor-real message, then as u^i does not store v^j, v^j is forwarded in a forward message to a node $w^k \in u^i.N$. Then $(u^i, w^k) \in E_e^{t+1}$ and $(v^j, w^k) \in E_i^{t+1}$ and u^i and v^j stay connected.
- v^j is stored in a probing message, then u^i forwards v^j to a node $w^k \in u^i.N$. Then $(u^i, w^k) \in E_e^{t+1}$ and $(v^j, w^k) \in E_i^{t+1}$ and u^i and v^j stay connected.

Note that if $u.m$ changes and a new virtual node u^j is simulated by u connectivity among all virtual nodes is preserved as the new virtual node u^j will only be connected to nodes u^0 and u^{i-1} are already connected to.

After establishing that connectivity is preserved, we define a time t_0 at which every message initially in $u^i.Ch$ for any node u^i has been received and also the responding messages have been delivered. Based on this we can show that eventually a state $s_{t'}$ at time $t' \geq t_0$ in the computation is reached such that $G_{list}^{t'}$ is weakly connected and stays weakly connected in any state s_t at time $t \geq t'$. In fact we show that for each edge (u^i, v^j) in E^{t_0} eventually u^i and v^j are connected in $G_{list}^{t'}$ and stay connected for ever after. For each edge (u^i, v^j) in E^{t_0} there are basically two options for u^i. Either v^j is stored in one of the internal variables or v^j is forwarded. This holds for all cases except that v^j is in a cycle-virtual message, then u^i responds by a respond-cycle-virtual message.

We first consider the case that v^j is eventually forwarded in a forward message. We assume w.l.o.g. that $h(v^j) > h(u^i)$. Then v^j is forwarded to a node $w^k \in u^i.N$ with $h(u^i) < h(w^k) < h(v^j)$. Thus there is a time t_1 such that a connecting path $p_1 = (u^i, w^k, v^j)$ in E^{t_1} exists. Now v^j is stored in $w^k.Ch$ in a forward message. And thus $x_1 = w^k$ again has two options to either keep v^j in an internal variable in every state after or to forward v^j eventually to another node x_2 with $h(x_1) < h(x_2) < h(v^j)$. Then v^j can only be forwarded $\mathcal{O}(n \log n)$ times. We denote the node v^j is forwarded to at the lth forwarding step by x_l. Let $l' = \mathcal{O}(n \log n)$ be the number of times v^j was forwarded.Then at time $t_{l'}$, we get a forwarding sequence of $fs(u^i, v^j) = (u^i = x_0, x_1, \cdots x_{l'}, v^j)$ with $x_{l+1} \in x_l.N$ at time t_l.

As already observed x_l either stores x_{l+1} in an internal variable forever or also eventually forwards it in a forward message. If x_{l+1} is forwarded we can construct a forwarding sequence $fs(x_l, x_{l+1})$ for each edge (x_l, x_{l+1}) as $x_l = x_{(l,l+1),0}, x_{(l,l+1),1}, \cdots x_{(l,l+1),o}, x_{l+1}$ with $h(x_l) < h(x_{(l,l+1),p-1}) <$

$h(w^{(i-1,i),p}) < h(x_{p+1})$ and $x_{(l,l+1),p} \in x_{(l,l+1),p-1}.N$ for every $p < o$. Inductively we can apply the same arguments and replace the resulting edges by further forwarding sequences, but as the number of nodes is finite eventually at some time t' we get a sequence $u^i = \tilde{x}_0, \tilde{x}_1, \tilde{x}_2, \cdots \tilde{x}_{\tilde{l}}, \tilde{x}_{\tilde{l}+1} = v^j$, where $\tilde{x}_{\tilde{j}-1}$ stores $\tilde{x}_{\tilde{j}}$ in an internal variable in every state after $t \geq t'$.

It remains to show that $(\tilde{x}_{\tilde{j}-1}, \tilde{x}_{\tilde{j}}) \in E^t_{list}$. If $\tilde{x}_{\tilde{j}} \in \left\{\tilde{x}_{\tilde{j}-1}.successor_v, \tilde{x}_{\tilde{j}-1}.predecessor_v\right\}$ obviously $(\tilde{x}_{\tilde{j}-1}, \tilde{x}_{\tilde{j}}) \in E^t_{list}$. If $\tilde{x}_{\tilde{j}} \in \left\{\tilde{x}_{\tilde{j}-1}.successor_r, \tilde{x}_{\tilde{j}-1}.predecessor_r\right\}$ then in the periodic action $\tilde{x}_{\tilde{j}-1}$ will introduce $\tilde{x}_{\tilde{j}}$ to $\tilde{x}_{\tilde{j}-1}.successor_v$ resp. $\tilde{x}_{\tilde{j}-1}.predecessor_v$. Thus we can replace $(\tilde{x}_{\tilde{j}-1}, \tilde{x}_{\tilde{j}})$ by another forwarding sequence $fs(\tilde{x}_{\tilde{j}-1}, \tilde{x}_{\tilde{j}})$ given by a sequence of virtual successors. If $\tilde{x}_{\tilde{j}} \in \left\{\tilde{x}_{\tilde{j}-1}.cycle_v, \tilde{x}_{\tilde{j}-1}.cycle_r, \tilde{x}^0_{\tilde{j}-1}, \cdots \tilde{x}^{(\tilde{x}_{\tilde{j}-1}).m}_{\tilde{j}-1}\right\}$, then $\tilde{x}_{\tilde{j}-1}$ will start a probing to $\tilde{x}_{\tilde{j}}$. Thus we can replace $(\tilde{x}_{\tilde{j}-1}, \tilde{x}_{\tilde{j}})$ by a probing sequence $ps(\tilde{x}_{\tilde{j}-1}, \tilde{x}_{\tilde{j}}) = (\tilde{x}_{\tilde{j}-1} = \tilde{x}_{(\tilde{j}-1,\tilde{j}),0}, \cdots \tilde{x}_{(\tilde{j}-1,\tilde{j}),\tilde{o}}, \tilde{x}_{\tilde{j}})$ with $h(\tilde{x}_{\tilde{j}-1}) < h(\tilde{x}_{(\tilde{j}-1,\tilde{j}),\tilde{p}-1}) < h(\tilde{x}_{(\tilde{j}-1,\tilde{j}),\tilde{p}}) < h(\tilde{x}_{\tilde{j}})$ with $\tilde{p} < \tilde{o}$ and $\tilde{x}_{(\tilde{j}-1,\tilde{j}),\tilde{p}} \in \tilde{x}_{(\tilde{j}-1,\tilde{j}),\tilde{p}-1}.N$. By applying these arguments inductively we get a sequence of virtual successors or predecessors connecting u^i and v^j at some time $t' \geq t_0$ such that no virtual successor or predecessor is forwarded in any state $t \geq t'$. Thus eventually u^i and v^j are connected in E^t_{list} and E^{t+1}_{list} for $t \geq t' \geq t_0$.

As we noted before this analysis holds for all cases except when v^j is in a cycle-virtual message in $u^i.Ch$. we can reduce this case to u^i being in a respond-cycle-virtual message in $v^j.Ch$. Then we can apply the aforementioned arguments. □

From Lemma 4.2.1 follows that there is a time t_0 such that all nodes are weakly connected in G^t_{list} with $t \geq t_0$ and stay weakly connected. Thus we can now show that $G^{t_0}_{list}$ eventually converges to a sorted list. To show this convergence we reuse the analysis presented to prove Theorem 3.2.2, where we showed the convergence property for P^{LIST}.

Lemma 4.2.2 *If the computation of $P^{RE-CHORD}$ reaches a state s_t $t \geq t_0$ where for some node u^i there are two edges $(u^i, v^j) \in E^t_{e-list}$ and $(u^i, w^k) \in E^t_{i-list}$ such that $h(u^i) < h(w^k) < h(v^j)$ (resp. $h(v^j) < h(w^k) < h(u^i)$) then this computation contains a later state $s_{t'}$ $t' > t$ with a edge $(u^i, w'^l) \in E^{t'}_{e-list}$ with $h(w'^l) \leq h(w^k)$ (resp. $h(w'^l) \leq h(w^k)$).*

Remind that this lemma claims that the edges $(u^i, u^i.successor_v)$ and $(u^i, u^i.predecessor_v)$ are shortened over time.

Proof. The proof is analogous to the proof of Lemma 3.2.2. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v^j)$ used in $P^{RE-CHORD}$ and the sets E^t_e by E^t_{e-list}, E^t_i by E^t_{i-list} and E^t by E^t_{list} as defined above. □

Lemma 4.2.3 *If the computation of $P^{RE-CHORD}$ reaches a state s_t $t \geq t_0$ where for a node u^i there are edges $(u^i, v^j) \in E^t_{e-list}$ and $(u^i, w^k) \in E^t_{i-list}$ with $h(u^i) < h(v^j) < h(w^k)$ (resp. $h(u^i) > h(v^j) > h(w^k)$) then the computation contains a later state $s_{t'}$ $t' > t$ where there is an edge $(w'^l, w^k) \in E^{t'}_{list}$ with $h(u^i) < h(w'^l) < h(w^k)$ (resp. $h(u^i) > h(w'^l) > h(w^k)$).*

Proof. The proof is analogous to the proof of Lemma 3.2.3. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v^j)$ used in $P^{RE-CHORD}$ and the sets E^t_e by E^t_{e-list}, E^t_i by E^t_{i-list} and E^t by E^t_{list}. □

Lemma 4.2.4 *If the computation of $P^{RE-CHORD}$ reaches a state s_t $t \geq t_0$ where for some nodes u^i, v^j and w^k such that $h(u^i) < h(w^k) < h(v^j)$ (resp. $h(v^j) < h(w^k) < h(u^i)$) there are edges $(u^i, v^j) \in E^t_{e-list}$ and $(w^k, u^i) \in E^t_{e-list}$ then the computation contains a later state $s_{t'}$ $t' > t$ where either some edge in $E^{t'}_{e-list}$ is shorter than in E^t_{e-list} or $(u^i, w^k) \in E^{t'}_{e-list}$.*

Proof. The proof is analogous to the proof of Lemma 3.2.4. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v^j)$ used in $P^{RE-CHORD}$ and the sets E^t_e by E^t_{e-list}, E^t_i by E^t_{i-list} and E^t by E^t_{list}. □

Lemma 4.2.5 *If the computation of $P^{RE-CHORD}$ reaches a state s_t $t \geq t_0$ where there is an edge $(u^i, v^j) \in E^t_{e-list}$ then the computation contains a later state $s_{t'}$ $t' \geq t$ where some edge in $E^{t'}_{e-list}$ is shorter than in E^t_{e-list} or $(v, u) \in E^{t'}_{e-list}$.*

Proof. The proof is analogous to the proof of Lemma 3.2.5. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v^j)$ used in $P^{RE-CHORD}$ and the sets E^t_e by E^t_{e-list}, E^t_i by E^t_{i-list} and E^t by E^t_{list}. □

Lemma 4.2.6 *If the computation of $P^{RE-CHORD}$ reaches a state s_t $t \geq t_0$ such that $(u^i, v^j) \in E^{t'}_{e-list} \Rightarrow (v^j, u^i) \in E^{t'}_{e-list}$ in every state $s_{t'}$ $t' \geq t$ after, then this computation contains a state s_{t^*} such that $E^{t^*}_{e-list}$ is strongly connected.*

Proof. The proof is analogous to the proof of Lemma 3.2.6. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v^j)$ used in $P^{RE-CHORD}$ and the sets E^t_e by E^t_{e-list}, E^t_i by E^t_{i-list} and E^t by E^t_{list}. □

Lemma 4.2.7 *If the computation of $P^{RE-CHORD}$ reaches a state s_t $t \geq t_1$ such that E^t_{e-list} is strongly connected and for every pair of nodes $(u^i, v^j) \in E^t_{e-list} \Rightarrow (v^j, u^i) \in E^t_{e-list}$ then this state is a solution for the sorted list problem and G^t_{list} is a sorted list.*

Proof. The proof is analogous to the proof of Lemma 3.2.7. We only substitute messages $m = (v)$ used in P^{LIST} by messages $m = (forward, v^j)$ used in $P^{RE-CHORD}$ and the sets E^t_e by E^t_{e-list}, E^t_i by E^t_{i-list} and E^t by E^t_{list}. □

We are now ready to show the correctness of Theorem 4.2.4. According to Lemma 4.2.1 G^t_{list} is weakly connected and stays weakly connected during the computation after some state s_{t_0}. By Lemma 4.2.2 and 4.2.3 follows that all edges in G^t_{list} are shortened over time. Note that as soon as a node receives a forward message there has to be a state s_{t_1} at time $t_1 \geq t_0$ with at least one edge in $E^{t_1}_{e-list}$. According to Lemma 4.2.4 eventually all edges in $E^{t_1}_{e-list}$ can not be shortened and either (u^i, v^j) and $(v^j, u^i) \in E^{t_2}_{e-list}$ at some state s_{t_2} at time $t_2 \geq t_1$ or a new edge is added to $E^{t_1}_{e-list}$. As for each node there are at most two edges in E_{e-list}, eventually all edges are added and a state s_{t_3} at time $t_3 \geq t_2$ is reached with $(u^i, v^j) \in E^{t3}_{e-list} \Rightarrow (v^j, u^i) \in E^{t_3}_{e-list}$. According to Lemma 4.2.5 there is a state s_{t_4} at time $t_4 \geq t_3$ such that in every state s_t at time $t \geq t_4$ $(u^i, v^j) \in E^t_{e-list} \Rightarrow (v^j, u^i) \in E^t_{e-list}$. Now, according to Lemma 4.2.6 the computation contains a later state s_{t_5} such that $E^{t_5}_{e-list}$ is strongly connected. Then by applying lemma 4.2.7 $G^{t_5}_{list}$ is a solution for the sorted-list problem.

The next theorem claims that the computation reaches a state, such that a cyclic list is a sub graph of the graph of explicit edges.

Theorem 4.2.5 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORD}$ then eventually the computation reaches a state s_t such that $G^t_{e-cycle}$ is a cyclic list.*

Proof. The proof is analogous to the proof of Theorem 3.3.4. We only substitute messages $m = (cycle, v)$and $m = (respond - cycle, v)$ used in $P^{SMALL-WORLD}$ by messages $m = (cycle - virtual, v^j)$and $m = (respond - cycle - virtual, v^j)$ used in $P^{RE-CHORD}$ and the sets $E^t_{e-cycle}$, $E^t_{i-cycle}$, E^t_{cycle} as defined for $P^{SMALL-WORLD}$ by $E^t_{e-cycle}$, $E^t_{i-cycle}$, E^t_{cycle} defined for $P^{RE-CHORD}$. □

The next theorem states that eventually a Re-Chord network is formed.

Theorem 4.2.6 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORD}$ then eventually the computation reaches a state s_t such that $G_r(G^t_e)$ is a Re-Chord network.*

Proof. From Theorem 4.2.5 follows that all virtual nodes form a sorted cyclic list at some state s_{t_0} and every state s_t with $t \geq t_0$. Then for each real node u^0 its virtual predecessor (resp. successor) $v^j = u^0.predecessor_v$ knows its real successor (resp. predecessor). $v^j.successor_r = v^j.successor_v = u^0$. Then if v^j is not a real node itself it will inform its virtual predecessor (resp. successor) about u^0. Following from this argument eventually all virtual nodes v^j between u^0 and $w^0 = predecessor_r(u^0)$ (resp. $w^0 = successor_r(u^0)$) are a aware of u^0 as their real successor $successor_r(v^j)$ (resp. predecessor $predecessor_r(v^j)$). As soon as a real node u^0 knows its real successor, i.e. $u^0.successor_r = successor_r(u^0)$ u^0 computes the correct number of virtual nodes it has to simulate, i.e. $u^0.m = m(u^0)$. Thus at this point all virtual nodes are created. Note that inserting a new virtual node u^i does not effect the self-stabilization process, as all its predecessor and successors are already stored in u^{i-1} and u^0. As soon as every virtual node u^i knows its real successor (resp. predecessor) $successor_r(u^i)$ (resp. predecessor $predecessor_r(u^i)$) we can apply the same arguments as for 4.2.5 to show that eventually for all virtual nodes $u^i.cycle_r = cycle_r(u^i)$. □

Then Theorem 4.2.3 follows immediately.

Closure

We will now show that for $P^{RE-CHORD}$ also the closure property holds.

Theorem 4.2.7 *If in an initial graph G with $G_r(G) \in RE-CHORD$ every nodes executes the protocol $P^{RE-CHORD}$ then each possible computation leads to a graph G' such that $G_r(G') \in RE-CHORD$.*

Proof. W.l.o.g we only consider edges $(u^i, v^j) \in E^t_e$ with $h(u^i) < h(v^j)$. Explicit edges in G^t_e do not change, as edges $(u^i, u^i.successor_v)$ only change if u^i receives a node w^k with $h(u^i) < h(w^k) < h(u^i.successor_v)$. As $u^i.successor_v = successor_v(u^i)$ (resp. $u^i.predecessor_v = predecessor_v(u^i)$) such a node w^k cannot exist.

Furthermore edges $(u^i, u^i.successor_r)$ only change if u^i receives a node w^0 with $h(u^i) < h(w^0) < h(u^i.successor_r)$. As $u^i.successor_r = successor_r(u^i)$ such a node w^0 cannot exist.

Edges $(u^i, u^i.cycle_v)$ with $h(u^i) < h(u^i.cycle_v)$ only change if u^i receives a node w^k with $h(w^k) > h(u^i.cycle_v)$ or $h(w^k) < h(u^i)$. As all virtual nodes form a cyclic list there cannot be a node w^k with

$h(w^k) < h(u^i)$ if $u^i.predecessor_v = nil$. There also cannot be a node w^k with $h(w^k) > h(u^i.cycle_v)$ as $u^i.cycle_v = cycle_v(u^i) = \text{argmax}\,\{h(v) : v \in V^+\}$.

Edges $(u^i, u^i.cycle_r)$ with $h(u^i) < h(u^i.cycle_r)$ only change if u^i receives a node w^0 with $h(w^0) > h(u^i.cycle_r)$ or $h(w^0) < h(u^i)$. As for all virtual node $u^i.successor_r = successor_r(u^i)$ such a node w^0 with $h(w^k) < h(u^i)$ cannot exist. There also cannot be a node w^0 with $h(w^0) > h(u^i.cycle_r)$ as $u^i.cycle_r = cycle_r(u^i) = \text{argmax}\,\{h(v) : v \in V\}$. □

4.2.8 Protocol $P^{RE-CHORDsync}$

In this section we adapt $P^{RE-CHORD}$ for the STSS model and present $P^{RE-CHORDsync}$. In this adapted protocol $P^{RE-CHORDsync}$ we reuse the same modifications for a synchronous setting as in the previously described protocols $P^{SMALL-WORLDsync}$ and $P^{LISTsync}$. In particular we collect all forward messages received in one round in a set $LIST$. We then sort all received identifiers and forward a node to its preceding neighbor in this sorting. Then a node u^i sends only $\mathcal{O}(1)$ forward messages to another node v^j. We use similar techniques to reduce the number of probing messages, which are collected in a set $Probing$. Furthermore we use introduction messages to avoid too many introductions by nodes that cannot be u^i's successor or predecessor. We also collect cycle-virtual messages in a set $Cycle$ to avoid cycle-virtual messages from nodes that should not miss a successor or predecessor. By this we can reduce the number of messages. For simplicity we assume that τ is true in every round.

We present an implementation in Pseudo code of $P^{RE-CHORDsync}$ in Algorithms 4.2.9- 4.2.16.

4.2.9 Analysis in the STSS model

In this section we analyze $P^{RE-CHORDsync}$ according to the STSS model. In particular we consider the stabilization time, stabilization work, the maintenance work and the work and time needed to recover from a single join or leave event.

Theorem 4.2.8 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{RE-CHORDsync}$ then the graph converges to a graph $G' \in RE-CHORD$ (Convergence) with a stabilization time of $\mathcal{O}(n \log n)$ and a stabilization work of $\mathcal{O}(n \log^3 n)$ w.h.p.. If in an initial graph $G \in RE-CHORD^*$ every node executes the protocol $P^{RE-CHORD}$ then each possible computation leads to a graph $G' \in RE-CHORD$ (Closure) with a maintenance work of $\mathcal{O}(\log^2 n)$. In a legal state it takes $\mathcal{O}(\log^2 n)$ rounds and $\mathcal{O}(\log^4 n)$ messages w.h.p. to recover and stabilize after a new real node joins or leaves the network.*

To prove this theorem we use the same formal definitions of the intermediate graphs and subsets of the edges like in the analysis of $P^{RE-CHORD}$.

Stabilization Time

We start by showing the stabilization time of the protocol $P^{RE-CHORDsync}$. Remind that in the $STSS$ model we execute all enabled actions in one round, i.e. all messages in $u^i.Ch$ are received by a node u^i.

Theorem 4.2.9 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORDsync}$ then the graph converges to a graph $G' \in RE-CHORD$ (Convergence) with a stabilization time of $\mathcal{O}(n \log n)$ w.h.p..*

Algorithm 4.2.9 $P^{RE-CHORDsync}$

message $m \in u^i.Ch \rightarrow$
$List = \emptyset$ ▷ Set of nodes that should be forwarded by forward messages
$Cycle = \emptyset$ ▷ Set of nodes that should be responded to by respond-cycle-virtual messages
$Probing = \emptyset$ ▷ Set of nodes that should be forwarded by probing messages
while $m \in u^i.Ch$ **do**
  if type=forward **then**
    List.insert(v^j)
  else if type=introduction **then**
    List.insert(v^j)
  else if type=probing **then**
    Probing.insert(v^j)
  else if type=cycle-virtual **then**
    Cycle.insert(v^j)
  else if type=respond-cycle-virtual **then**
    respond-cycle-virtual(v^j)
  else if type=cycle-neighbor-real **then**
    cycle-neighbor-real(v^j)
send-probes()
send-cycle()
send-list()
$\tau \rightarrow$
introduction()
probing()

Algorithm 4.2.10 $P^{RE-CHORDsync}$-SEND-LIST()

if $List \neq \emptyset$ **then**
  Sort all ids v^j in $List \cup u^i.successor_v \cup u^i.predecessor_v \cup u^i.successor_r \cup u^i.predecessor_r$ according to their position $h(v)$, such that $h(v^{-k}) < h(v^{-(k-1)}) < \cdots < h(v^{-1}) < h(u^i) < h(v^1) < \cdots < h(v^{k'-1}) < h(v^{k'})$
  $u^i.successor_v = v^1$
  $u^i.successor_r = \text{argmin}\left\{h(v^l) : l > 0 \wedge v^l \text{ is real}\right\}$
  for j=1 to k'-1 **do**
    Send message $m' = (forward, v^{j+1})$ to v^j
    if v^j is received by an introduction message **then**
      send message $m' = (forward, v^{j-1})$ to v^j
  $u^i.predecessor_v = v^{-1}$
  $u^i.predecessor_r = \text{argmax}\left\{h(v^l) : l < 0 \wedge v^l \text{ is real}\right\}$
  for j=1 to k-1 **do**
    Send message $m' = (forward, v^{-(j+1)})$ to v^{-j}
    if v^j is received by an introduction message **then**
      send message $m' = (forward, v^{-(j-1)})$ to v^{-j}

Algorithm 4.2.11 $P^{RE-CHORDsync}$-SEND-CYCLE()

if $Cycle \neq \emptyset$ **then**
Sort all ids v^j in $Cycle$ according to their position $h(v)$, such that $h(v^{-k}) < h(v^{-(k-1)}) < \cdots < h(v^{-1}) < h(u^i) < h(v^1) < \cdots < h(v^{k'-1}) < h(v^{k'})$
for $j = 1$ to $k' - 1$ **do**
Send message $m' = (forward, v^{j+1})$ to v^j
if $\exists w^l : h(w) > h(v^{k'}) \wedge w^l \in u^i.N$ **then**
$w^l = \text{argmin} \left\{ h(w') : h(w') > h(v^{k'}) \wedge w' \in u^i.N \right\}$
send message $m = (forward, w^l)$ to $v^{k'}$
else if $\exists w^l : h(w^l) < h(v) \wedge w^l \in u^i.N$ **then**
$w^l = \text{argmin} \left\{ h(w') : h(w') < h(v) \wedge w' \in u^i.N \right\}$
send message $m = (respond - cycle - virtual, w^l)$ to v
else
send message $m = (respond - cycle - virtual, u^i)$ to v
for $j = 1$ to $k - 1$ **do**
Send message $m' = (forward, v^{-(j+1)}$ to v^{-j}
if $\exists w^l : h(w) < h(v^k) \wedge w^l \in u^i.N$ **then**
$w^l = \text{argmax} \left\{ h(w') : h(w') < h(v^k) \wedge w' \in u^i.N \right\}$
send message $m = (forward, w^l)$ to $v^{k'}$
else if $\exists w^l : h(w^l) > h(v) \wedge w^l \in u^i.N$ **then**
$w^l = \text{argmax} \left\{ h(w') : h(w') > h(v) \wedge w' \in u^i.N \right\}$
send message $m = (respond - cycle - virtual, w^l)$ to v
else
send message $m = (respond - cycle - virtual, u^i)$ to v

Algorithm 4.2.12 $P^{RE-CHORDsync}$-SEND-PROBES()

if $Probing \neq \emptyset$ **then**
Sort all ids v^j in $Probing \cup u^i.successor_v \cup u^i.predecessor_v \cup \left\{ u^0, \cdots u^{u.m} \right\}$ according to their position $h(v)$, such that $h(v^{-k}) < h(v^{-(k-1)}) < \cdots < h(v^{-1}) < h(u^i) < h(v^1) < \cdots < h(v^{k'-1}) < h(v^{k'})$
for $j = 1$ to $k' - 1$ **do**
if $v^{j+1} \notin u^i.successor_v, u^i.predecessor_v, u^0, \cdots u^{u.m}$ **then**
Send message $m' = (probing, v^{j+1})$ to v^j
List.insert(v^1)
for $j = 1$ to $k - 1$ **do**
if $v^{-(j+1)} \notin u^i.successor_v, u^i.predecessor_v, u^0, \cdots u^{u.m}$ **then**
Send message $m' = (probing, v^{-(j+1)}$ to v^{-j}
List.insert(v^{-1})

Algorithm 4.2.13 $P^{RE-CHORDsync}$-RESPOND-CYCLE-VIRTUAL(v)

if $u^i.predecessor_v = nil \wedge (h(v) > h(u^i.cycle_v) > h(u^i) \vee (u^i.cycle_v = nil \wedge h(v) > h(u^i)))$ **then**
  if $u^i.cycle_v \neq nil$ **then**
    $w = u^i.cycle_v$
  $u^i.cycle_v = v$
  $List.insert(w)$
else if $u^i.successor_v = nil \wedge (h(v) < h(u^i.cycle_v) < h(u^i) \vee (u^i.cycle_v = nil \wedge h(v) < h(u^i)))$ **then**
  if $u^i.cycle_v \neq nil$ **then**
    $w = u^i.cycle_v$
  $u^i.cycle_v = v$
  $List.insert(w)$
else
  $List.insert(v)$

Algorithm 4.2.14 $P^{RE-CHORDsync}$-CYCLE-NEIGHBOR-REAL(v)

if v is real **then**
  if $u^i.predecessor_r = nil \wedge (h(v) > h(u^i.cycle_r) > h(u^i) \vee (u^i.cycle_r = nil \wedge h(v) > h(u^i)))$ **then**
    if $u^i.cycle_r \neq nil$ **then**
      $w = u^i.cycle_r$
    $u^i.cycle_r = v$
    List.insert(w)
  else if $u^i.successor_r = nil \wedge (h(v) < h(u^i.cycle_r) < h(u^i) \vee (u^i.cycle_r = nil \wedge h(v) < h(u^i)))$ **then**
    if $u^i.cycle_r \neq nil$ **then**
      $w = u^i.cycle_r$
    $u^i.cycle_r = v$
    List.insert(w)
  else
    List.insert(v)
else
  List.insert(v)

Algorithm 4.2.15 $P^{RE-CHORDsync}$-INTRODUCTION()

if $h(u^i.predecessor_v) \neq nil$ **then**
  send message $m = (introduction, u^i)$ to $u^i.predecessor_v$
else
  send message $m = (cycle - virtual, u^i)$ to $u^i.cycle_v$
if $h(u^i.successor_v) \neq nil$ **then**
  send message $m = (introduction, u^i)$ to $u^i.successor_v$
else
  send message $m = (cycle - virtual, u^i)$ to $u^i.cycle_v$
if $h(u^i.predecessor_r) \neq nil$ **then**
  send message $m = (forward, u^i.predecessor_r)$ to $u^i.predecessor_v$
  if u^i is not real **then**
    if $u^i.successor_v \neq nil$ **then**
      send message $m = (forward, u^i.predecessor_r)$ to $u^i.successor_v$
    else
      send message $m = (cycle - neighbor - real, u^i.predecessor_r)$ to $u^i.cycle_v$
else
  send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.predecessor_v$
  if u^i is not real **then**
    send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.successor_v$
if $h(u^i.successor_r) \neq nil$ **then**
  send message $m = (forward, u^i.successor_r)$ to $u^i.successor_v$
  if u^i is not real **then**
    if $u^i.predecessor_v \neq nil$ **then**
      send message $m = (forward, u^i.successor_r)$ to $u^i.predecessor_v$
    else
      send message $m = (cycle - neighbor - real, u^i.successor_r)$ to $u^i.cycle_v$
else
  send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.successor_v$
  if u^i is not real **then**
    send message $m = (cycle - neighbor - real, u^i.cycle_r)$ to $u^i.predecessor_v$

Algorithm 4.2.16 $P^{RE-CHORDsync}$-PROBING()

if $\exists u^j : j > i$ **then**
  if $h(u^j) < h(u^i.predecessor_v)$ **then**
    send message $m = (probing, u^j)$ to $u^i.predecessor_v$
  else if $h(u^j) > h(u^i.successor_v)$ **then**
    send message $m = (probing, u^j)$ to $u^i.successor_v$
if $u^i.cycle_v \neq nil \wedge h(u^i.cycle_v) < h(u^i.predecessor_v)$ **then**
  send message $m = (probing, u^i.cycle_v)$ to $u^i.predecessor_v$
else if $u^i.cycle_v \neq nil \wedge h(u^i.cycle_v) > h(u^i.successor_v)$ **then**
  send message $m = (probing, u^i.cycle_v)$ to $u^i.successor_v$
if $u^i.cycle_r \neq nil \wedge h(u^i.cycle_r) < h(u^i.predecessor_v)$ **then**
  send message $m = (probing, u^i.cycle_r)$ to $u^i.predecessor_v$
else if $u^i.cycle_r \neq nil \wedge h(u^i.cycle_r) > h(u^i.successor_v)$ **then**
  send message $m = (probing, u^i.cycle_r)$ to $u^i.successor_v$

Like in the analysis of $P^{RE-CHORD}$ in the ATSS model we prove the theorem by showing some intermediate steps. We start by showing that in $\mathcal{O}(n \log n)$ rounds w.h.p. all nodes are weakly connected in E_{list} and stay weakly connected.

Theorem 4.2.10 *After $\mathcal{O}(n \log n)$ rounds w.h.p. all virtual nodes are weakly connected in $G = (V^+, E_{list})$ and stay weakly connected.*

Proof. We can use the same arguments as in the proof of Lemma 4.2.1 to show that all virtual nodes are weakly connected and stay weakly connected in $G = (V^+, E)$. We proceed by showing that nodes that are connected in E_{list} stay connected. If $(u^i, v^j) \in E^t_{list}$ then either $v^j \in \{u^i.successor_v, u^i.predecessor_v\}$ or $m = (forward, v^j) \in u^i.Ch$ or $m = (introduction, v^j) \in u^i.Ch$ in either case if $(u^i, v^j) \notin E^{t+1}_{list}$ then v^j is forwarded in the send-list() action. Then according to the protocol $P^{RE-CHORDsync}$ there is a path $(u^i = x_0, \cdots x_k = v^j)$ with $(x_i, x_{i+1}) \in E^{t+1}_{list}$ and u^i and v^j stay connected in E^{t+1}_{list}.

It remains to show that for all $(u^i, v^j) \in E^0 - E^0_{list}$ after $t = \mathcal{O}(n \log n)$ rounds u^i and v^j are connected in E^t_{list}. W.l.o.g. we assume $h(v_j) > h(u_i)$.

If $(u^i, v^j) \in E^0 - E^0_{list}$ then $v^j \in \{u^i.successor_r, u^i.cycle_v, u^i.cycle_r\}$ or $m = (type, v^j) \in u^i.Ch$ of type probing, cycle-virtual, respond-cycle-virtual or cycle-neighbor-real. If $v^j = u^i.successor_r$, then u^i will send a message $m = (forward, v^j)$ to $u^i.successor_v$ and thus after one round u^i and v^j are connected in E^t_{list}.

We now show that all virtual siblings are connected in E^t_{list} after at most $\mathcal{O}(n \log n)$ rounds. We prove this by induction. Let u^i, u^{i+1} be a pair of virtual siblings, such that in $[u^i, u^{i+1}]$ is no other pair of virtual siblings v^j, v^{j+1}. W.l.o.g. we assume $h(u^i < h(u^{i+1}))$. Then the probing from u^i to u^{i+1} can only proceed by forwarding the probing message to virtual or real successors or to other probed nodes. Let w^k be a node receiving the $m = (probing, u^{i+1})$ probing initiated by u^i. Then let Δ be the number of nodes w' with $h(w) < h(w') < h(u^{i+1})$. Then $m = (probing, u^{i+1})$ can only be forwarded at most $\Delta - 1$ further times before an edge $(w^k, u^{i+1}) \in E^t_{list}$ is created. If $m = (probing, u^{i+1})$ is forwarded to a virtual successorw'^l of w^k then by our previous analysis we know that w^k and w'^l are connected and stay connected in E^t_{list}. If $m = (probing, w'^l) \in w^k.Ch$ then $m = (probing, w'^l)$ can also be forwarded at

most $\Delta - 1$ times. By applying this argument inductively we get a probing sequence $ps(u^i, u^{i+1})$ for the pair u^i, u^{i+1} with $p = (u^i = x_0, x_1 \cdots, u^{i+1})$ after $\mathcal{O}(n \log n)$ rounds w.h.p. (as $\Delta = \mathcal{O}(n \log n)$ w.h.p.), such that $x_{i+1} \in \{x_i.successor_v, x_i.successor_r\}$. Then after $\mathcal{O}(n \log n)$ rounds w.h.p. u^i and u^{i+1} are connected in E^t_{list}.

In the induction step we assume that for a pair u^i, u^{i+1} all virtual siblings v^j, v^{j+1} in $[u^i, u^{i+1}]$ are connected in E^t_{list} after $\mathcal{O}(n \log n)$ rounds w.h.p.. Let again w^k be a node receiving the $m = (probing, u^{i+1})$ probing initiated by u^i. Then let Δ be the number of nodes w'^l with $h(w^k) < h(w'^l) < h(u^{i+1})$. Then $m = (probing, u^{i+1})$ can only be forwarded at most $\Delta - 1$ further times before an edge $(w^k, u^{i+1}) \in E^t_{list}$ is created. If $m = (probing, u^{i+1})$ is forwarded to a virtual successor w'^l of w^k then by our previous analysis we know that w^k and w'^l are connected and stay connected in E^t_{list}. If $m = (probing, u^{i+1})$ is forwarded to a virtual sibling w^l of w^k then by the induction hypothesis we know that w^k and w^l will be connected and stay connected in E^t_{list}.

If $m = (probing, w'^l) \in w^k.Ch$ then $m = (probing, w'^l)$ can also be forwarded at most $\Delta - 1$ times. By applying this argument inductively we get a probing sequence $ps(u^i, u^{i+1})$ for the pair u^i, u^{i+1} with $p = (u^i = x_0, x_1 \cdots, u^{i+1})$ after $\mathcal{O}(n \log n)$ rounds w.h.p. (as $\Delta = \mathcal{O}(n \log n)$ w.h.p.), such that x_{i+1} is connected to and stays connected to x_i in E^t_{list}. Then after $\mathcal{O}(n \log n)$ rounds w.h.p. u^i and u^{i+1} are connected in E^t_{list}.
If a node u^i receives a probing message $m = (probing, v^j) \in u^i.Ch$ (w.l.o.g. $h(v_j) > h(u_i)$) it either forwards v^j by another probing message to a node $w \in u^i.Probing$ or $w \in \{u^i.successor_v, u^0, \cdots, u^{u.m}\}$ with $h(u^i) < h(w) < h(v^j)$ or if v^j cannot be forwarded an edge $(u^i, v^j) \in E^t_{list}$ is created.

Let Δ be the number of nodes w , such that $h(u^i) < h(w) < h(v^j)$. Then obviously $\Delta = \mathcal{O}(n \log n)$ and $m = (probing, v^j)$ can only be forwarded Δ times. If u^i forwards $m = (probing, v^j)$ to a node w with $m = (probing, w) \in u^i.Ch$ then $m = (probing, v^j)$ and $m = (probing, w)$ can only be forwarded at most $\Delta - 1$ times. If u^i forwards $m = (probing, v^j)$ to a node w with $w = u^k \in \{u^0, \cdots, u^{u.m}\}$ then $m = (probing, v^j)$ can only be forwarded at most $\Delta - 1$ times and by our previous analysis we know that u^i and u^k will be connected as virtual siblings in E^t_{list} after $\mathcal{O}(n \log n)$ rounds. If u^i forwards $m = (probing, v^j)$ to a node w with $w = u^i.successor_v$ then we already know that u^i and w are connected and stay connected in E^t_{list} and $m = (probing, v^j)$ can only be forwarded at most $\Delta - 1$ times. By applying these arguments inductively we get a probing sequence $ps(u^i, v^j)$ for the pair u^i, v^j with $p = (u^i = x_0, x_1 \cdots, v^j)$ after $\mathcal{O}(n \log n)$ rounds w.h.p. (as $\Delta = \mathcal{O}(n \log n)$ w.h.p.), such that x_{i+1} is connected to and stays connected to x_i in E^t_{list}. Then after $\mathcal{O}(n \log n)$ rounds w.h.p. u^i and v^j are connected in E^t_{list}.

If $v^j = u^i.cycle_v$ or $v^j = u^i.cycle_r$ then u^i will send a probing message $m = (probing, v^j)$ in the probing() action. Thus we can reduce this case to the case of receiving a probing message from v^j. If $m = (respond - cycle - virtual, v^j) \in u^i.Ch$, u^i either sets $u^i.cycle_v = v^j$ or treats m as a forward message. Thus we can reduce it to one of the cases considered above. If $m = (cycle - neighbor - real, v^j) \in u^i.Ch$, u^i either sets $u^i.cycle_r = v^j$ or treats m as a forward message. Thus we can also reduce it to one of the cases considered above. If $m = (cycle - virtual, v^j) \in u^i.Ch$, u^i will send a message $m = (respond - cycle - virtual, w^k)$ containing a node $w^k \in u^i.N$. Thus by our analysis we know that u^i and w^k and v^j and w^k will be connected in E^t_{list} after $\mathcal{O}(n \log n)$ rounds. Then also u^i and v^j are connected. □

Theorem 4.2.11 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORDsync}$ then the graph converges to a graph G' such that G'_{e-list} is a sorted list after $\mathcal{O}(n \log n)$ rounds w.h.p..*

Proof. By Theorem 4.2.10 we know that after $\mathcal{O}(n \log n)$ rounds w.h.p. a state s_{t_0} is reached such that all virtual nodes are weakly connected and stay weakly connected in E^t_{list} for $t \geq t_0$. We use a similar technique to form a sorted list in $P^{RE-CHORDsync}$ as in P^{LIST}, i.e. we sort the nodes locally according to their positions. The only difference is that we allow shortcuts by forwarding an identifier to a virtual sibling. We therefore can simply apply Theorem 3.2.5, that provides an upper bound of $\mathcal{O}(n \log n)$ rounds w.h.p., as $|V^+| = \mathcal{O}(n \log n)$ w.h.p.. □

Theorem 4.2.12 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORDsync}$ then the graph converges to a graph G' such that $G'_{e-cycle}$ is a cyclic list after $\mathcal{O}(n \log n)$ rounds w.h.p..*

Proof. According to Theorem 4.2.11 the computation reaches a state s_{t_0} such that all virtual nodes form a sorted list after at most $\mathcal{O}(n \log n)$ rounds w.h.p.. Thus only the virtual nodes with the minimal v_{min} and maximal v_{max} position are missing a virtual predecessor or successor. Then these nodes send in the introduction() action cycle-virtual messages to to their current cycle nodes $w = v_{min}.cycle_v$ and $w' = v_{max}.cycle_v$. W.l.o.g. we consider only the case of v_{min} as the analysis for v_{max} is symmetric. Then w is either v_{max} or w sends a respond-cycle-virtual message to v_{min} containing a node x with $h(x) > h(w)$. Then this action is repeated. If $|V^+| = \mathcal{O}(n \log n)$ w.h.p. $v_{min}.cycle_v$ is updated at most $\mathcal{O}(n \log n)$ times before $v_{min}.cycle_v = v_{max}$. □

Theorem 4.2.13 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{RE-CHORDsync}$ then the graph converges to a graph G' such that $G_r(G')$ is a Re-Chord network after $\mathcal{O}(n \log n)$ rounds w.h.p..*

Proof. According to Theorem 4.2.12 the computation reaches a state s_{t_0} such that all virtual nodes form a cyclic list after at most $\mathcal{O}(n \log n)$ rounds w.h.p.. in every state s_t for $t \geq t_0$. Thus for each node it holds that u^i $u^i.successor_v = successor_v(u^i)$ and $u^i.predecessor_v = predecessor_v(u^i)$ and $u^i.cycle_v = cycle_v(u^i)$. Then also for every real node u^0 $u^0.successor_v.predecessor_r = u^0$ and $u^0.predecessor_v.successor_r = u^0$. Let $v^j = u^0.successor_v$ then in the introduction() action v^j will inform its virtual successor about u^0. As there are at most $\log n$ virtual nodes between a pair of consecutive real nodes w.h.p. after $\mathcal{O}(\log n)$ each virtual node knows its real predecessor and by the same arguments the real successor and real cycle node. Then for each node u^i $u^i.successor_r = successor_r(u^i)$ and $u^i.predecessor_r = predecessor_r(u^i)$ and $u^i.cycle_r = cycle_r(u^i)$. □

Then Theorem 4.2.9 follows immediately. We conclude the analysis by showing the Closure property in the STSS model.

Theorem 4.2.14 *If in an initial graph G with $G_r(G) \in RE - CHORD$ every nodes executes the protocol $P^{RE-CHORDsync}$ then each possible computation leads to a graph G' such that $G_r(G') \in RE - CHORD$.*

Proof. The proof is the same as in the asynchronous setting for $P^{RE-CHORD}$. □

Stabilization work

Theorem 4.2.15 *If in an initial graph $G \in IT$ every node executes the protocol $P^{RE-CHORD}$ then each possible computation leads to a graph G' with $G_r(G') \in RE - CHORD$ with a stabilization work of $\mathcal{O}(n \log^3 n)$ w.h.p. per node.*

Proof. In each round a virtual node sends $\mathcal{O}(1)$ introduction messages to its virtual successor and predecessor. A virtual node also sends $\mathcal{O}(1)$ cycle-virtual messages and probing messages in each round. Furthermore a virtual node only receives $\mathcal{O}(n \log n)$ introduction messages, as a node u^i only receives an introduction message once from a node v^j, if v^j is not its virtual successor or predecessor. If u^i receives introduction messages from two nodes v^j and w^k and $h(v^j) < h(w^k) < h(u^i)$ (resp. $h(v^j) > h(w^k) > h(u^i)$) then u^i will send a message $m = (forward, w^k)$ to v^j thus v^j updates its virtual successor (resp. predecessor) and will not send an introduction message to u^i again. As cycle-real-neighbor messages are only sent to virtual successor or predecessors it follows that a node also only receives $\mathcal{O}(n \log n)$ cycle-real-neighbor messages.

A virtual node also only receives at most $\mathcal{O}(n \log n)$ cycle-virtual messages, as a node u^i only receives a cycle-virtual message once from a node v^j which is not the current node with the minimal or maximal position sending a cycle-virtual message to u^i. If u^i receives cycle-virtual messages from two nodes v^j and w^k and $h(v^j) < h(w^k) < h(u^i)$ (resp. $h(v^j) > h(w^k) > h(u^i)$) then u^i will send a message $m = (forward, v^j)$ to w^k thus w^k updates its virtual predecessor (resp. predecessor) and will not send an cycle-virtual message to u^i again. Then u^i also only sends $\mathcal{O}(n \log n)$ respond-cycle-virtual messages. A virtual node u^i also only receives $\mathcal{O}(n \log n)$ respond-cycle-virtual messages.

A virtual node u^i sends at most $\mathcal{O}(1)$ probing messages to another node v^j. A probing message is only sent to a virtual successor or predecessor, a virtual sibling or another probed node. As we have already seen a node can be the virtual successor or predecessor for at most $\mathcal{O}(n \log n)$ nodes during the stabilization. Thus a node u^i can receive at most $\mathcal{O}(n \log n)$ probing messages by nodes storing u^i as the virtual predecessor or successor. Furthermore a node u^i has $\mathcal{O}(\log n)$ virtual siblings w.h.p. Thus during the stabilization u^i receives at most $\mathcal{O}(n \log^2 n)$ probing messages by its virtual siblings. If there are probing messages containing u^i, u^i might also receive probing messages. As we have already seen there are at most $\mathcal{O}(n \log n)$ nodes w with $w.cycle_v = u^i$ during the stabilization, thus there are also at most $\mathcal{O}(n \log n)$ probes containing u^i initiated by these nodes. Additionally u^{i-1} initiates probes to u^i in each round resulting in $\mathcal{O}(n \log n)$ probes containing u^i. Thus during the stabilization there are at most $\mathcal{O}(n \log n)$ probes containing u^i then u^i also receives only $\mathcal{O}(n \log^2 n)$ probing messages. As a node u^i only initiates probing messages to one virtual sibling u^{i+1} and its virtual cycle node $u^i.cycle_v$ a node u^i sends in total $\mathcal{O}(n \log^2 n)$ probing messages resulting from the $\mathcal{O}(1)$ messages initiated in each round and the $\mathcal{O}(n \log^2 n)$ probing messages received.

To bound the number of forward messages we follow the ideas presented in the proof of 3.2.10. We show that a node u receives at most $\mathcal{O}(n \log n)$ forward messages with nodes v^j, such that $h(u^i) < h(v^j)$. There are three cases in which u^i receives such a message. Either u^i sent an introduction message to a node w with $h(v^j) < h(w)$ and received v^j as a response or v^j is sent to u^i by a node w' with $h(w') < h(u^i) < h(v^j)$ that simply forwards v^j to a closer node according to its local sorting or receives v^j as a response to a cycle-virtual message previously send by u^i.

The first and last case can happen at most once. As soon as u^i receives v^j it will never introduce itself to such a node w with $h(v^j) < h(w)$. If u^i receives v^j as a response to cycle-virtual message, then u^i was missing a virtual successor and now received a virtual successor and thus stops sending cycle-virtual messages. So we can focus on the second case. Let's assume u^i receives a_k forward messages containing the node w_k from nodes w' with $h(w') < h(u^i)$ where the w_ks are sorted in descending order $h(w_k) > h(w_{k+1})$. The a_k messages containing w_k can have the three aforementioned sources. Either w_k was stored initially in a node w' with $h(w') < h(u^i)$ or such a node w' received w_k as a response to an introduction or cycle-virtual message. We therefore split a_k in to two components. Let b_k be the number of times u^i receives w_k because initially there is an edge $(w', w_k) \in E^0$ and over time w_k is

forwarded to u^i. And let c_k be the number of times u^i receives w_k, because w' received w_k after an introduction or virtual-cycle message. We first can observe that $\sum_{i=1}^{l} b_i = \mathcal{O}(n \log n)$. If a node w' initially has edges (w', w_k) and (y, w_l) with $l > k$ then this w^k will never be forwarded to u^i, as it will be forwarded to a node v with $h(u^i) < h(w_l) \leq h(v) < h(w_k)$. So there have to be $\sum_{k=1} b_k$ different nodes w' with $h(w') < h(u^i)$. There can be at most $\mathcal{O}(n \log n)$ such nodes w.h.p..

To give a bound on $\sum_{k=1} c_k$ we have to determine how many different w_ks a node w' can receive as a response to an introduction or cycle-virtual message. If a node receives w_k it cannot have an edge (w', w_l) with $l > k$ as otherwise w' would not send an introduction message to a node w with $h(w) > h(w_k) > h(w_l)$ also w' would not miss a virtual successor and send a cycle-virtual message. So w' receives w_l after w_k, but before w' forwards w_k to a node v with $h(w') < h(v) \leq h(u^i)$, because otherwise w' would not send an introduction message to a node w'' with $h(w'') > h(w_l) > h(v)$. So either w' has edges to w_l and w_k at the same time and then forwards w_k to v' with $h(v') > h(u^i)$ and this w_k is never forwarded to u^i or w_k is forwarded to such a node v' before receiving w_l. So each node w' receives at most one of the w_ks as a response. Then $\sum_{i=k} c_k = \mathcal{O}(n \log n)$ w.h.p..

If a node receives at most $\mathcal{O}(n \log n)$ forward messages w.h.p. it will also send at most $\mathcal{O}(n \log n)$ forward messages w.h.p.. Summing up we get that each virtual node sends or receives at most $\mathcal{O}(n \log^2 n)$ messages w.h.p.. As each real node simulates $\mathcal{O}(\log n)$ virtual nodes w.h.p. each real node send or receives $\mathcal{O}(n \log^3 n)$ in total.

□

Maintenance work

Theorem 4.2.16 *If in an initial graph $G \in RE - CHORD^*$ every nodes executes the protocol $P^{RE-CHORD}$ then each possible computation leads to a graph $G' \in RE-CHORD$ with a maintenance work of $\mathcal{O}(\log^2 n)$.*

Proof. Each virtual node sends to and receives from its virtual successor and predecessor $\mathcal{O}(1)$ introduction messages. In a stable state no forward messages are sent or received. Each virtual node sends to and receives from its virtual successor and predecessor $\mathcal{O}(1)$ cycle-real-neighbor messages. As only the virtual nodes with a minimal or maximal position send one cycle-virtual message a node can only receive $\mathcal{O}(1)$ such messages, then also only $\mathcal{O}(1)$ respond-cycle-virtual messages are sent and received. A node only receives probing messages by its virtual successor and predecessor or its virtual siblings so in total $\mathcal{O}(\log n)$ w.h.p. or at most one probing message for each probing message destined for the node itself. There are at most $\mathcal{O}(1)$ probes initiated to a virtual node in each round. Summing up a virtual node receives and sends at most $\mathcal{O}(\log n)$ messages in each round w.h.p.. Then a real node sends and receives at most $\mathcal{O}(\log^2 n)$ messages in one round in a stable state w.h.p.. □

Single Join and Leave Event

In this section we examine the number of steps needed for a network to recover to its stable state when a node joins or leaves the network.

When a node joins the network, it is initially connected with an arbitrary real node already in the network and it is integrated to its stable position by the protocol $P^{RE-CHORDsync}$.

Theorem 4.2.17 *In a legal state it takes $\mathcal{O}(\log^2 n)$ rounds and $\mathcal{O}(\log^4 n)$ messages w.h.p. to recover and stabilize after a new real node joins or leaves the network.*

Proof. If a node u joins the network and is connected to a real node v already in the network then, according to $P^{RE-CHORDsync}$, u determines $u.m$ and simulates the corresponding virtual nodes u^i. Thus for at least one u^i $u^i.successor_v = v$ or $u^i.predecessor_v = v$. Then u^i introduces itself to v. Then either v stores u^i in its internal variables $v.successor_v$, $v.predecessor_v$ or $v.cycle_v$ or forwards u^i. Then after $\mathcal{O}(\log n)$ rounds u^i will reach its correct virtual successor or predecessor. Then after $\mathcal{O}(\log n)$ further rounds u^i will know its other correct real successor w^0 and predecessor $w'^0$and its virtual successor and predecessor (or cycle nodes). Then there is another virtual sibling u^j such that w^0 or w'^0 are its real predecessor or successor. Thus after $\mathcal{O}(\log n)$ further rounds by the same arguments also u^j will know its correct real and virtual neighbors. As u simulates $\mathcal{O}(\log n)$ virtual nodes it takes $\mathcal{O}(\log^2 n)$ rounds to integrate u and all its virtual nodes.

A new node u_i sends at most $\mathcal{O}(1)$ forward messages as it is only (initially) connected to at most $\mathcal{O}(1)$ nodes v^j that are not its virtual successor or predecessor. As soon as u^i updates its virtual successor and predecessor it is connected to its correct virtual successor or predecessor, and then only has to forward the old values once. In each round a joining node u_i sends $\mathcal{O}(1)$ introduction, cycle-virtual messages and cycle-real-neighbor messages. As seen in the analysis of the maintenance work each node receives and sends at most $\mathcal{O}(\log n)$ probing messages in one round. This argument still holds as each u^i can only be the virtual successor or predecessor of $\mathcal{O}(1)$ nodes in one round. Then a joining real node u has to send and receive $\mathcal{O}(\log^4 n)$ messages until a joining node is integrated. A node v^j already in the network is a virtual successor or predecessor of at most $\mathcal{O}(1)$ nodes u^i thus v^j only receives $\mathcal{O}(1)$ messages in one round additional to its maintenance work while all u^i are integrated.

If a node u leaves the network also all its virtual nodes leave the network. So there are $\mathcal{O}(\log n)$ gaps in the cyclic list. If a gap exists because a virtual node u^i $i > 0$ left, then this gap will be closed in $\mathcal{O}(\log n)$ rounds, as the virtual successor and predecessor w^j of u^i now will take their real successor or predecessor v^0 as also their virtual successor or predecessor. Then this real node v^0 will forward w^j to its correct virtual predecessor or successor. As there are w.h.p. at most $\mathcal{O}(\log n)$ virtual nodes between two real nodes after $\mathcal{O}(\log n)$ rounds w.h.p. $w^j.successor_v = successor_v(w^j)$ or $w^j.predecessor_v = predecessor_v(w^j)$ and the gap is closed.

If a gap exists because the real node u^0 left, then its virtual successor and predecessor no longer have a virtual predecessor or successor and will set their virtual cycle node to the node with the maximal or minimal position they know. As all other gaps are closed after $\mathcal{O}(\log n)$ rounds w.h.p. this start-point of this cycle edge will be updated along a routing path thus $\mathcal{O}(\log n)$ times w.h.p. and the gap is closed.

In the first case a real node v^0 receives at most $\mathcal{O}(1)$ additional introduction messages and sends $\mathcal{O}(1)$ additional forward messages. In the second case at most $\mathcal{O}(1)$ nodes (u's virtual successor and predecessor) send a cycle-virtual message, then a node v_j can receive at most $\mathcal{O}(1)$ additional cycle-virtual messages in one round. Then each virtual node still sends or receives $\mathcal{O}(\log n)$ messages in each round. Thus a real node v sends and receives at most $\mathcal{O}(\log^3 n)$ messages until a stable state is reached. □

4.3 CONE-DHT: A distributed self-stabilizing algorithm for a heterogeneous storage system

4.3.1 Introduction

In this section we consider the problem of designing distributed protocols for a dynamic heterogeneous storage system

In this section we present a self-stabilizing overlay network for a distributed heterogeneous storage system based on the data assignment presented in [72].

We state the following requirements on our solution: *Fair load balancing*: every node with x% of the available capacity gets x% of the data. *Space efficiency*: Each node stores at most $\mathcal{O}(|\text{data assigned to the node}| + \log n)$ information. *Routing efficiency*: There is a routing strategy that allows efficient routing in at most $\mathcal{O}(\log n)$ hops. *Low degree*: The degree of each node is limited by $\mathcal{O}(\log n)$. Furthermore we require an algorithm that builds the target network topology in a *self-stabilizing* manner, i.e., any weakly connected network $G = (V, E)$ is eventually transformed into a network so that a (specified) subset of the explicit edges forms the target network topology (*convergence*) and remains stable as long as no node joins or leaves (*closure*).

4.3.2 Our contribution

We present a protocol P^{CONE} solving the overlay problem $CONE$. The ideas of the protocol P^{CONE} and the CONE-DHT overlay network, the results for the correctness in the ATSS model in Theorem 4.3.5 and the case of a joining or leaving node given in Theorem 4.3.21 and Theorem 4.3.22 are based on results published in [37] which is a joint work with my colleague Andreas Koutsopoulos and our supervisor Christian Scheideler:

Sebastian Kniesburges, Andreas Koutsopoulos and Christian Scheideler, CONE-DHT: A distributed self-stabilizing algorithm for a heterogeneous storage system, 27th International Symposium on Distributed Computing (DISC 2013)

The results for the stabilization time in the STSS model in Theorem 4.3.13, stabilization time in Theorem 4.3.18 and maintenance work in Theorem 4.3.19 are presented for the first time.

We present a self-stabilizing algorithm that organizes a set of heterogeneous nodes in an overlay network such that each data element can be efficiently assigned to the node responsible for it. We use the scheme described in [72] (which gives us good load balancing) as our data management scheme and present a distributed protocol for the overlay network, which is efficient in terms of message complexity and information storage and moreover works in a self-stabilizing manner. The overlay network efficiently supports the basic operations of a heterogeneous storage system, such as the joining or leaving of a node, changing the capacity of a node, as well as searching, deleting and inserting a data element. In fact we show the following results:

There is a topological self-stabilizing protocol for maintaining a heterogeneous storage system that achieves fair load-balancing, space efficiency and routing efficiency, while each node has a degree of $\mathcal{O}(\log n)$ w.h.p.. The protocol P^{CONE} is correct in the ATSS and STSS model. In the STSS model we show a stabilization time of $\mathcal{O}(n)$, a stabilization work of $\mathcal{O}(n^2 + Dn \log n)$ w.h.p. if there are D data items stored in the network, a maintenance work of $\mathcal{O}(r + \log^2 n)$ w.h.p. if a node stores data items supervised by r reference nodes. A single join, leave or capacity change of a node can be handled in $\mathcal{O}(\log n)$ rounds and needs $\mathcal{O}(\log^3 n)$ work. The data operations can be handled in $O(\log n)$ time in a stable system.

4.3.3 The original CONE-Hashing

Before we present our solution, we first give some more details on the original approach that we call CONE-Hashing [72]. In [72] the authors present a centralized solution for a heterogeneous storage system

in which the nodes are of different capacities. We denote the capacity of a node u as $c(u)$. We use a hash function $h : V \mapsto [0,1)$ that assigns to each node a hash value. A data element of the data set D is also hashed by a hash function $g : D \mapsto [0,1)$. W.l.o.g. we assume that all hash values and capacities are distinct.

According to [72] each node has a height function $H_u^f(x) = \frac{1}{c_u} f(|x - h(u)|)$, which determines which data is assigned to the node, where f is a monotonic increasing function. A node is *responsible* for those elements d with $H_u^f(g(d)) = \min_{v \in V}\{H_v^f(g(d))\}$, i.e. d is assigned to u. We denote by $R(u) = \{x \in [0,1) : H_u^f(x) = \min_{v \in V}\{H_v^f(x)\}\}$ the *responsibility range* of u (see Figure 4.5). Note that $R(u)$ can consist of several intervals in $[0,1)$.

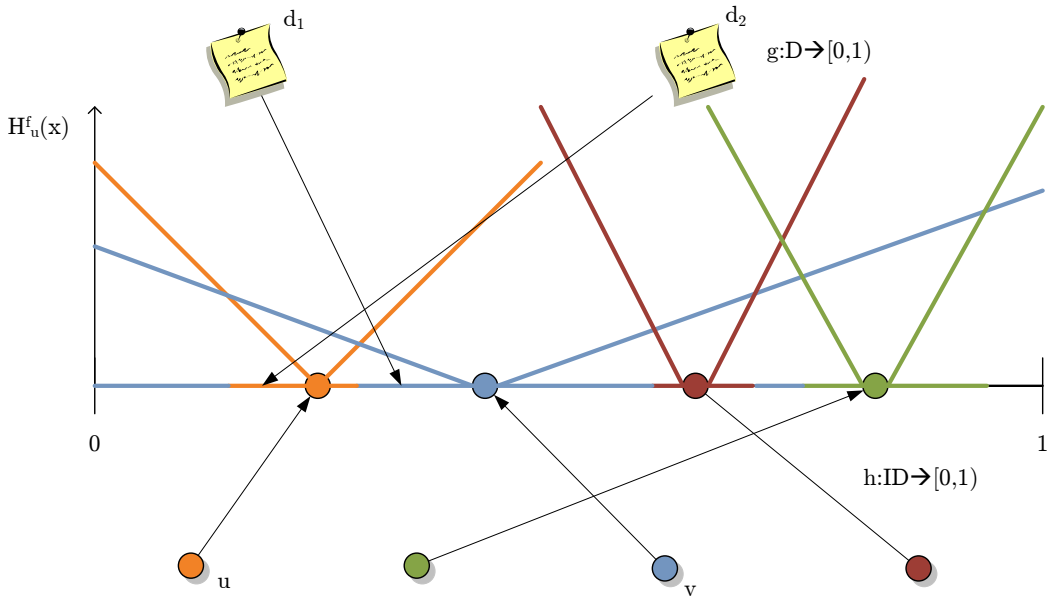


Figure 4.5: Responsibilities in CONE

In the original paper [72], the authors considered two cases of height functions, one of linear form $H_u^{lin}(x) = \frac{1}{c(u)}|x - h(u)|$ and of logarithmic form $H_u^{log}(x) = \frac{1}{c(u)}(-log(|1 - (x - h(u))|)$. For these height functions the following results were shown by the authors [72]:

Theorem 4.3.1 *A data element d is assigned to a node u with probability $\frac{c(u)}{\sum_{v \in V} c(v) - c(u)}$ for linear height functions $H_u^{lin}(x)$ and with probability $\frac{c(u)}{\sum_{v \in V} c(v)}$ for logarithmic height functions $H_u^{log}(x)$. Thus in expectation fair load balancing can be achieved by using a logarithmic height function $H_u^{log}(x)$.*

The CONE-Hashing supports the following operations for a data element d or a node v:

- *Search(d)*: Returns the node u such that $g(d) \in R(u)$.
- *Insert(d)*: d is assigned to the node returned by $Search(d)$.
- *Delete(d)*: d is removed from the node returned by $Search(d)$.

- *Join(v)*: For all $u \in V$ the responsibility ranges $R(u)$ are updated and data elements d, with $g(d) \in R(v)$ are moved to v.
- *Leave(v)*: For all $u \in V$ the responsibility ranges $R(u)$ are updated and data elements d assigned to v are moved to nodes w such that $g(d) \in R(w)$.
- *CapacityChange(v)*: For all $u \in V$ the responsibility ranges $R(u)$ are updated and data elements d not assigned to v, but with $g(d) \in R(v)$ are moved to v while data elements d' assigned to v but with $g(d) \in R(w)$ are moved to nodes w.

In [72] the authors further present a data structure to efficiently support the described operations in a centralized approach. For their data structure they showed that there is an algorithm that determines for a data element d the corresponding node u with $g(d) \in R(u)$ in expected time $\mathcal{O}(\log n)$. The used data structure has a size of $\mathcal{O}(n)$ and the joining, leaving and the capacity change of a node can be handled efficiently.

In the following we show that CONE-Hashing can also be realized by using a distributed data structure called CONE-DHT. We present a suitable topology on the node set V that supports an efficient determination of the responsibility ranges $R(u)$ for each node u . The topology supports an efficient *Search(d)* algorithm, i.e. for an *Search(d)* query inserted at an arbitrary node w, the node v with $g(d) \in R(v)$ should be found. Furthermore a *Join(v)*, *Leave(v)* and *CapacityChange(v)* operation does not lead to a high amount of data movements, (i.e. no more than the data now assigned to v or no longer assigned to v should be moved) or a high amount of structural changes (i.e. changes in the topology built on V).

4.3.4 The *CONE*-DHT

In order to built the CONE-DHT we introduce the *CONE*-overlay network, which can support efficiently a heterogeneous storage system.

The network layer

For the determination of the edge set of a CONE-network, we need following definitions, with respect to a node u:

- $succ_1^+(u) = \text{argmin}\{h(v) : h(v) > h(u) \wedge c(v) > c(u)\}$ is the next node at the right of u with larger capacity, and we call it the first larger successor of u. Building upon this, we define recursively the i-th larger successor of u as: $succ_i^+(u) = succ_1^+(succ_{i-1}^+(u)), \forall i > 1$, and the union of all larger successors as $S^+(u) = \bigcup_i succ_i^+(u)$.
- The first larger predecessor of u is defined as: $pred_1^+(u) = \text{argmax}\{h(v) : h(v) < h(u) \wedge c(v) > c(u)\}$ i.e. the next node at the left of u with larger capacity. The i-th larger predecessor of u is: $pred_i^+(u) = pred_1^+(pred_{i-1}^+(u)), \forall i > 1$, and the union of all larger predecessors as $P^+(u) = \bigcup_i pred_i^+(u)$.
- We also define the set of the smaller successors of u, $S^-(u)$, as the set of all nodes v, with $u = pred_1^+(v)$, and the set of the smaller predecessors of u, $P^-(u)$ as the set of all nodes v, such that $u = succ_1^+(v)$.

- We further define the successor of a node u $succ(u) = \text{argmin}\{h(v) : h(v) > h(u)\}$ and the predecessor of and node u $pred(u) = \text{argmax}\{h(v) : h(v) < h(u)\}$.
- We define also the neighborhood set of u as $N(u) = S^+(u) \cup P^+(u) \cup S^-(u) \cup P^-(u)$.

Now we can define the CONE-network in the following way.

Definition 4.3.1 *A graph $G = (V, E)$ is a* CONE-network *if V is the set of hosts in the network and $E = \{(u, v) : v \in S^+(u) \cup P^+(u) \cup S^-(u) \cup P^-(u)\}$.*

In other words, a node u maintains connections to each node u, if there does not exist another node w with larger capacity than $c(w) > c(v)$ between v and u (see Figure 4.6). We will prove that this graph is sufficient for maintaining a heterogeneous storage network in a self-stabilizing manner and also that in this graph the degree is bounded logarithmically w.h.p..

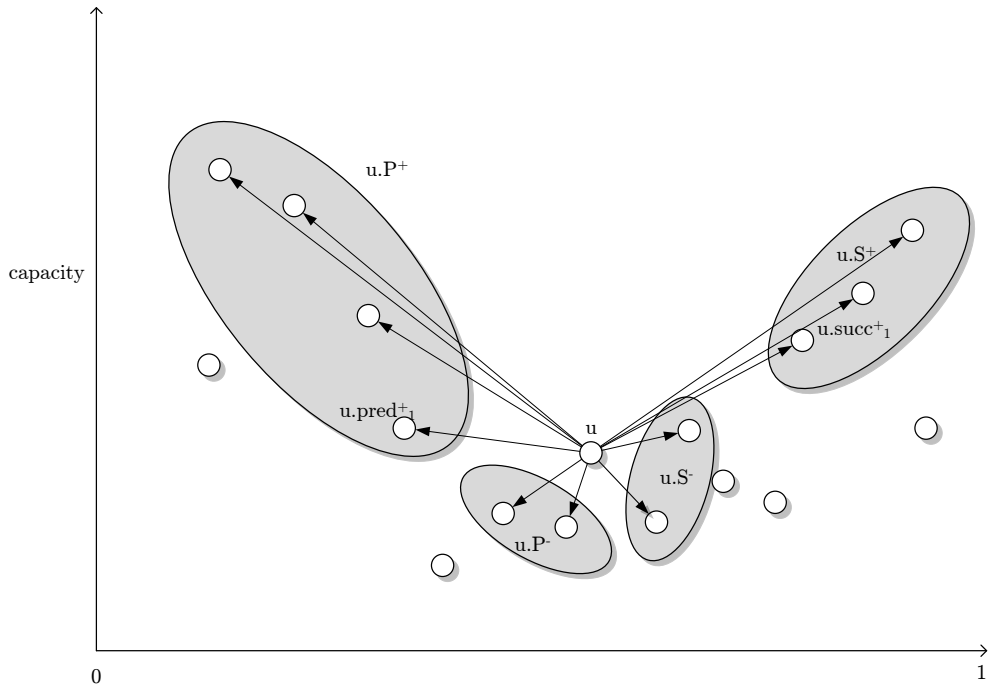


Figure 4.6: The CONE-network from u's perspective

The data management layer

We discussed above how the data is assigned to the different nodes. That is the assignment strategy we use for data in the CONE-network.

In order to understand how the various data operations are realized in the network, we have to describe how each node maintains the knowledge about the data it has, as well as the intervals it is responsible for. In fact in our CONE-DHT a node u does not know for which intervals it is responsible for, but there always is another node v that can decide if a data item d should be stored at node u. It turns out that in

order for a data item to be forwarded to the correct node, which is responsible for storing it, it suffices to contact the closest node (in terms of hash value) from the left to the data item's hash value. That is because then, if the CONE-network has been established, this node (for example node v in Figure 4.7) is aware of the responsible node u for this data item. We call the interval between $h(v)$ and the hash value of v's closest right node I_v. We say that v is *supervising* I_v. We show the following theorem.

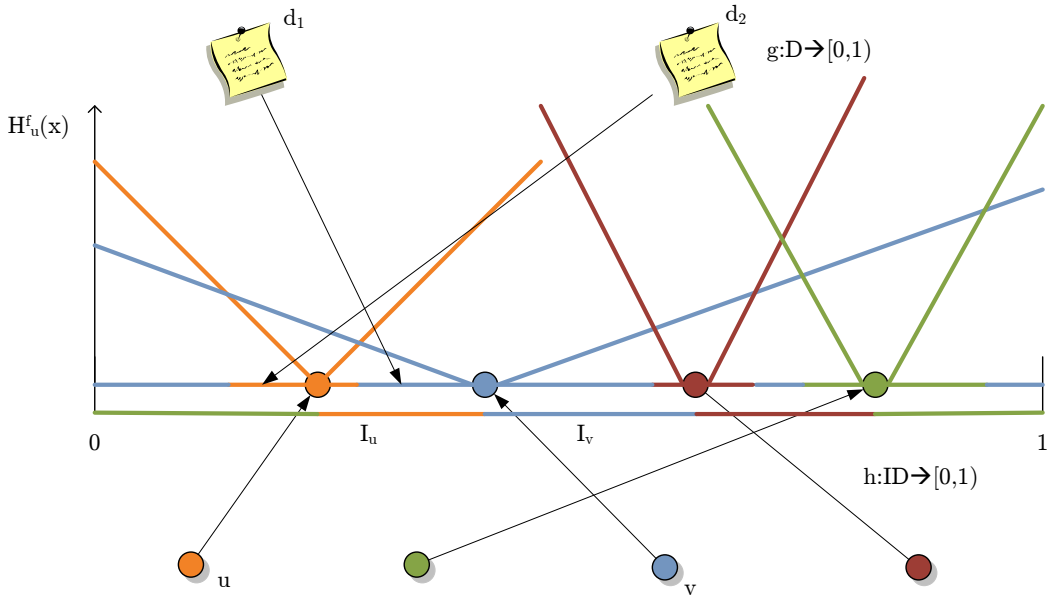


Figure 4.7: Responsibilities in CONE-DHT with supervised intervals

Theorem 4.3.2 *In a CONE-network a node u knows all the nodes v with $R(v) \cap I_u \neq \emptyset$.*

Proof. We need to show that all these nodes v with $R(v) \cap I_u \neq \emptyset$ are in $S^+(u) \cup P^+(u) \cup S^-(u) \cup P^-(u)$.

W.l.o.g. let us consider only the case of $h(v) > h(u)$, i.e. $v \in S^+(u) \cup S^-(u)$. We assume that there is a node w with $h(u) < h(w)$ that has a responsible interval in u's supervised interval and is not in $S^+(u)$ or $S^-(u)$. If w is not in $S^-(u)$ or $S^+(u)$ there must be at least one node w' with a larger capacity $c(w') > c(w)$ which is closer to u than w with $h(u) < h(w') < h(w)$. Then $\forall x < h(v) \in [0,1)$ it holds that $|x - h(w')| < |x - h(w)| \implies f(|x - h(w')|) < f(|x - h(w)|)$ for some height function $H^f_u(x)$, since f is increasing. Moreover, since $c(w') > c(w)$ we can show $\frac{1}{c(w')} f(|x - h(w')|) < \frac{1}{c(w)} f(|x - h(w)|) \implies H^f_{w'}(x) < H^f_w(x) \forall x < h(v) \in [0,1)$, so w' dominates w for those points x. And since $h(u) < h(w')$, it cannot be that w is responsible for an interval in I_u, since in that region w is dominated (at least) by w'. This contradicts our assumption about the existence of such a node w. □

So, the nodes store their data in the following way. If a node u has a data item d that falls into one of its responsible intervals, it stores in addition to this item a reference to the node v that supervises this interval. A node also stores the sub-interval it is responsible for in which the data item d falls. In case the data item is not stored at the correct node, v can resolve the conflict when contacted by u. Therefore u has to contact the reference node v periodically.

A node has operations for inserting, deleting and searching a data item in the CONE-network.

Let us focus on the search operation $Search(d)$ for a data item. Again we first search for the node u supervising the interval d falls into. Then u contacts the node responsible for it. We use a greedy routing scheme to find correct node v with $g(d) \in I_v$. If a search request wants to reach some position $g(d) = x$ in $[0, 1)$, and the request is currently at node u, then u forwards the search operation to the node w in $N(u)$ that is closest to x and $x > h(w) > h(u)$ or $x < h(w) < h(u)$, until the node supervising x is reached. Then this node will forward the request to the responsible node. A more formal definition of the greedy routing follows:

Definition 4.3.2 *The CONE Greedy routing strategy is defined as follows: If operation op is to be executed at position x in $[0, 1]$ and op is currently at node u, then u forwards op to the node v such that $v = argmax\{h(w) : w \in N(u) \cup \{u\} \wedge h(w) < x\}$ if $x > h(u)$ or u forwards op to the node v such that $v = argmin\{h(w) : w \in N(u) \cup \{u\} \wedge h(w) > x\}$ if $x < h(u)$. If $h(v) = h(u)$ and $x > h(u)$, then $x \in I_u$ and u forwards op to the node responsible for the sub-interval containing x. If $h(v) = h(u)$ and $x < h(u)$ then u forwards op to $pred(u)$ as x is in its supervised interval.*

In that way we can route to the responsible node and then get an answer whether the data item is found or not, and so the searching is realized. The delete operation $Delete(d)$ for a data item d can be realized in the same way, only that when the item is found, it is also deleted from the responsible node. An insert operation $Insert(d)$ follows a similar procedure, with the difference that when the responsible node is found, the data item is stored by it together with a reference to the supervising node.

Structural Properties of a Cone Network

In this section we show that the degree of a node in a stable CONE-network is bounded by $\mathcal{O}(\log n)$ w.h.p, and hence the information stored by each node (i.e the number of nodes which it maintains contact to, $|E_e(u)|$) is bounded by $\mathcal{O}(\log n + |$amount of data stored in a node$|)$ w.h.p..

First we show following lemma:

Lemma 4.3.1 *In a stable CONE network for each $u \in V$, $|S^+(u)|$ and $|P^+(u)|$ are in $\mathcal{O}(\log n)$ w.h.p.*

Proof. For an arbitrary $u \in V$ let $W = S^+(u) \cup \{u\} = \{w_0, w_1, w_2 \cdots w_k\}$ and let W be sorted by ids in ascending order, such that $h(w_i) < h(w_{i+1})$ for all $1 \leq i < k$. Furthermore, let $\hat{W}(w_i) = \{w \in V : h(w) > h(w_i) \wedge c(w) > c(w_i)\}$ be the set of all nodes with larger positions and larger capacities than w_i.

So, the determination of W and so $S^+(u)$ is done by continuously choosing the correct w_i out of $\hat{W}(w_{i-1})$, when $w_1, w_2, ...w_{i-1}$ are already chosen. In this process, each time a w_i is determined, the number of nodes from which w_{i+1} can be chosen is getting smaller, since the nodes at the left of w_i as well as the nodes with smaller capacity than w_i can be excluded. We call the choice of w_i *good*, if $|\hat{W}(w_{i-1})| > 2|\hat{W}(w_i)|$, i.e. the number of remaining nodes in $\hat{W}(w_i)$ is (at least) halved. Let $|\hat{W}(w_0)| = k = \mathcal{O}(n)$. Since the position $h(u)$ for each node u is assigned uniformly at random, we can easily see that $Pr[w_i$*is a good choice*$] = \frac{1}{2}, \forall i \geq 1$. Then after a sequence of i choices that contains $\log k$ good choices the remaining set $\hat{W}(w_i)$ is the empty set. Thus there can not be more than $\log k$ good choices in any sequence of choices.

So, what we have now is a random experiment, that is described by the random variable l, that is equal to the number of choices we must make, until we manage to have made $\log k$ good ones. Then the

random variable l follows the negative binomial distribution. In order to bound the value of l from above we apply the following tail bound for negative binomial distributed random variables derived by using a Chernoff bound:

Let Y have the negative binomial distribution with parameters s and p, i.e. with probability p there is a success and Y is equal to the number of trials needed for s successes. Pick $\delta \in [0,1]$. Then $Pr[Y > \frac{s}{(1-\delta)p}] \leq exp(\frac{-\delta^2 s}{3(1-\delta)})$

We apply this claim with $p = \frac{1}{2}$ and $s = \log k$ and we pick $\delta = \frac{7}{8}$. Then $Pr[Y > 16 \log k] \leq \exp(\frac{-\delta^2 2 \log k}{3(1-\delta)}) < k^{-2}$. Thus with probability at least $1 - k^{-2}$, $l = \mathcal{O}(\log k)$. As $k = \mathcal{O}(n)$ also $l = \mathcal{O}(\log n)$. □

Lemma 4.3.2 *In a stable CONE network for each $u \in V$, $E[|S^-(u)|]$ and $E[|P^-(u)|]$ are $\mathcal{O}(1)$ and $|S^-(u)|$ and $|P^-(u)|$ are $\mathcal{O}(\log n)$ w.h.p..*

Proof. W.l.o.g. we consider only $E[|S^-(u)|]$ and $|S^-(u)|$ in the proof. For each node v being in $S^-(u)$ it holds that $u = pred_1^+(v)$. Since each node has at most one $pred_1^+$, $\sum_{v \in V} S^-(v) = n$. Then for a node u, $E[S^-(u)] = 1$.

Now we consider the second part of the statement. Let v be the direct right neighbor of u, i.e. the first (from the left) node in $S^-(u)$ ($S^-(u)[1]$). Then we can observe that every node in $S^-(u)$ (except v) must be in $S^+(v)$. Let us assume that a node w is in $S^+(v)$ but not in $S^-(u)$, then there must be another node $y : h(u) < h(v) < h(y) < h(w)$ and $c(y) > c(w)$, such that $y \in S^-(u)$. But then y would be also in $S^+(v)$ instead of w. Then $S^-(u)/\{v\} \subseteq S^+(v)$, but we already showed that $|S^+(v)| = \mathcal{O}(\log n)$ w.h.p., from which follows that $|S^-(u)| = \mathcal{O}(\log n)$ w.h.p.. □

Combining the two lemmas we get the following theorem.

Theorem 4.3.3 *The degree of each node in a stable CONE network is $\mathcal{O}(\log n)$ w.h.p.*

Additionally to the nodes in $S^+(u)$, $S^-(u)$, $P^+(u)$ and $P^-(u)$ that lead to the degree of $\mathcal{O}(\log n)$ w.h.p. a node u only stores references to nodes supervising the intervals it is responsible for, where it actually stores data. A node u stores at most one reference and one interval for each data item.

Theorem 4.3.4 *In a stable CONE network each node stores at most $\mathcal{O}(\log n + |$amount of data$|)$ information w.h.p.*

Once the CONE network G^{CONE} is set up, it can be used as a heterogeneous storage system supporting inserting, deleting and searching for data. The CONE Greedy routing implies the following bound on the diameter:

Lemma 4.3.3 *CONE Greedy routing takes on a stable CONE network w.h.p. no more than a logarithmic number of steps, i.e. the diameter of a CONE network is $\mathcal{O}(\log n)$ w.h.p..*

Proof. This follows directly from Lemma 4.3.1, where we showed that each node u has w.h.p. a logarithmic number of nodes in $P^+(u)$ and $S^+(u)$, which means each node has a logarithmic distance to the node with the greatest capacity w.h.p., and vice versa, which means that the node with the greatest capacity has logarithmic distance to every node in the network. The proof for the CONE Greedy routing follows from a generalization of this observation. If an operation *op* with position $x \in I_u$ is currently at node v w.l.o.g. we assume $h(u) > h(v)$, then *op* is forwarded at most $\mathcal{O}(\log n)$ times w.h.p. to a node w such that $w \in S^+(v)$ and further $\mathcal{O}(\log n)$ times w.h.p. (along nodes in $S^+(u)$) from w to u. □

4.3.5 Formal definition

Now we define the problem we solve in this paper in the previously introduced notation. We provide a protocol P^{CONE} that solves the overlay problem *CONE* and is topologically self-stabilizing.

In order to give a formal definition of the edges in E_e and in E_i we firstly describe which internal variables are stored in each node u, i.e. which edges exist in E_e:

- $u.S^+$ a set of nodes v with $h(v) > h(u) \wedge c(v) > c(u)$ such that $\forall w \in u.N : h(v) > h(w) > h(u) \implies c(v) > c(w)$. All nodes in $u.S^+$ can be responsible in I_u.
- $u.succ_1^+ = \text{argmin}\,\{h(v) : v \in u.S^+\}$: The first node to the right with a larger capacity than u
- $u.P^+$ a set of nodes v with $h(v) < h(u) \wedge c(v) > c(u)$ such that $\forall w \in u.N : h(v) < h(w) < h(u) \implies c(v) > c(w)$. All nodes in $u.P^+$ can be responsible in I_u.
- $u.pred_1^+ = \text{argmax}\,\{h(v) : v \in u.P^+\}$: The first node to the left with a larger capacity than u
- $u.S^-$ a set of nodes v with $h(v) > h(u) \wedge c(v) < c(u)$ such that $\forall w \in u.N : h(v) > h(w) > h(u) \implies c(v) > c(w)$. All nodes in $u.S^-$ can be responsible in I_u.
- $u.P^-$ a set of nodes v with $h(v) < h(u) \wedge c(v) < c(u)$ such that $\forall w \in u.N : h(v) < h(w) < h(u) \implies c(v) > c(w)$. For all nodes in $u.P^-$ $v.succ_1^+ = u$.
- $u.S^* = \left\{u.S^- \cup \left\{u.succ_1^+\right\}\right\}$: the set of right neighbors that u communicates with. We assume that the nodes are stored in ascending order so that $h(u.S^*[i]) < h(u.S^*[i+1])$. If there are k nodes in $u.S^*$ then $u.S^*[k] = u.succ_1^+$.
- $u.P^* = \left\{u.P^- \cup \left\{u.pred_1^+\right\}\right\}$: the set of left neighbors that u communicates with. We assume that the nodes are stored in descending order so that $h(u.P^*[i]) > h(u.P^*[i+1])$ If there are k nodes in $u.P^*$ then $u.P^*[k] = u.pred_1^+$.
- *u.DS* the data set, containing all intervals $u.DS[i] = [a, b]$, for which u is responsible and stores actual data $u.DS[i].data$. Additionally for each interval a reference $u.DS[i].ref$ to the supervising node is stored

Additionally each node stores the following variables :

- $u.\tau$: the timer predicate that is periodically true
- $u.I_u$: the interval between u and the successor of u. u is supervising $u.I_u$.
- *u.Ch*: The channel for incoming messages.

The content of a message is given by the ids of some nodes and a specification of the message type. So different type of messages can trigger different actions. Thus a message m is of the following form $m = (type, ids)$.

- *forward*: By this message a node introduces itself to other nodes or other nodes to each other, if they should be connected.
- *list-update*: This type of message is used to update the P^+ and S^+ list in nodes in $u.S^-$ and $u.P^-$.

- *check-interval*: By this message a node checks whether it is responsible for a certain interval [a,b] by asking the supervising node.
- *update-interval*: By this message a supervising node informs a responsible node about the intervals it is responsible for.
- *forward-data*: This type of message is used to route data in the network.
- *store-data*: This type of message is used to tell a node to store the data it is responsible for.

In the following we define when an assignment of the variables is valid and how the sets of initial topologies IT and the goal topologies $CONE$ look like.

Definition 4.3.3 *We define a valid state as an assignment of values to the internal variables of all nodes so that the definition of the variables is not violated, e.g.* $u.S^+$ *contains no nodes* w *with* $h(w) < h(u)$ *or* $c(w) < c(u)$ *or* $h(u) < h(v) < h(w)$ *and* $c(v) > c(w)$ *for any* $v \in u.N$*. Note that an invalid assignment can be locally repaired immediately. Thus we assume in the following w.o.l.g. that initially and in every later state the assignment is valid for every node.*

Definition 4.3.4 *Let* E_e *and* E_i *be defined as described in Chapter 2 according to the definition of internal variables. Then the set of initial topologies is given by:*

$$IT = \{G = (V, E = E_e \cup E_i) : G \text{ is weakly connected}\}$$

Definition 4.3.5 *Let* E_e *and* E_i *be defined as described in Chapter 2 according to the definition of internal variables. Then the set of target topologies is given by:*

$$CONE = \{G = (V, E) : G_e \text{ is a CONE-network}\}$$

4.3.6 Protocol P^{CONE}

In this section we give a description of our algorithm. The algorithm is a protocol that each node executes based on its own node and channel state. The protocol contains periodic actions that are executed if the timer predicate τ is true and actions that are executed if the node receives a message m.

In order for a node u to maintain valid lists $(u.S^+, u.P^+)$, it makes a periodic check of its lists with its neighbors in $u.S^-, u.P^-$, where the lists are compared, so that inconsistencies are repaired. Moreover a node checks whether $u.S^-/u.P^-$ are valid and introduces to every node in $u.S^-/u.P^-$ to them their closest larger right/left neighbors (from u's perspective).

Unnecessary (for these lists) nodes are forwarded. Forwarding means that node u sends a node $v : h(v) > h(u)$ (resp. $h(v) < h(u)$) to the node $w' = \text{argmax}\{h(w) : w \in u.S^* \wedge h(w) < h(v)\}$ (resp. $w' = \text{argmin}\{h(w) : w \in u.P^* \wedge h(w) > h(v)\}$) by a message $m = (forward, v)$ to w'. The idea is to forward nodes closer to their correct position, so that a sorted list is formed.

In the periodic actions each node introduces itself to its successor and predecessor $u.S^*[1]$ and $u.P^*[1]$ by a message $m = (forward, u)$. Also each pair of nodes (v, w) in $u.P^*$ and $u.S^*$ with consecutive positions $h(v)$ and $h(w)$ are introduced to each other. A node u also introduces the nodes $u.succ_1^+$ and $u.pred_1^+$ to each other by messages of type $forward$. By this a triangle is formed by edges $(u, u.pred_1^+), (u, u.succ_1^+), (u.succ_1^+, u.pred_1^+)$ (see Figures 4.8).

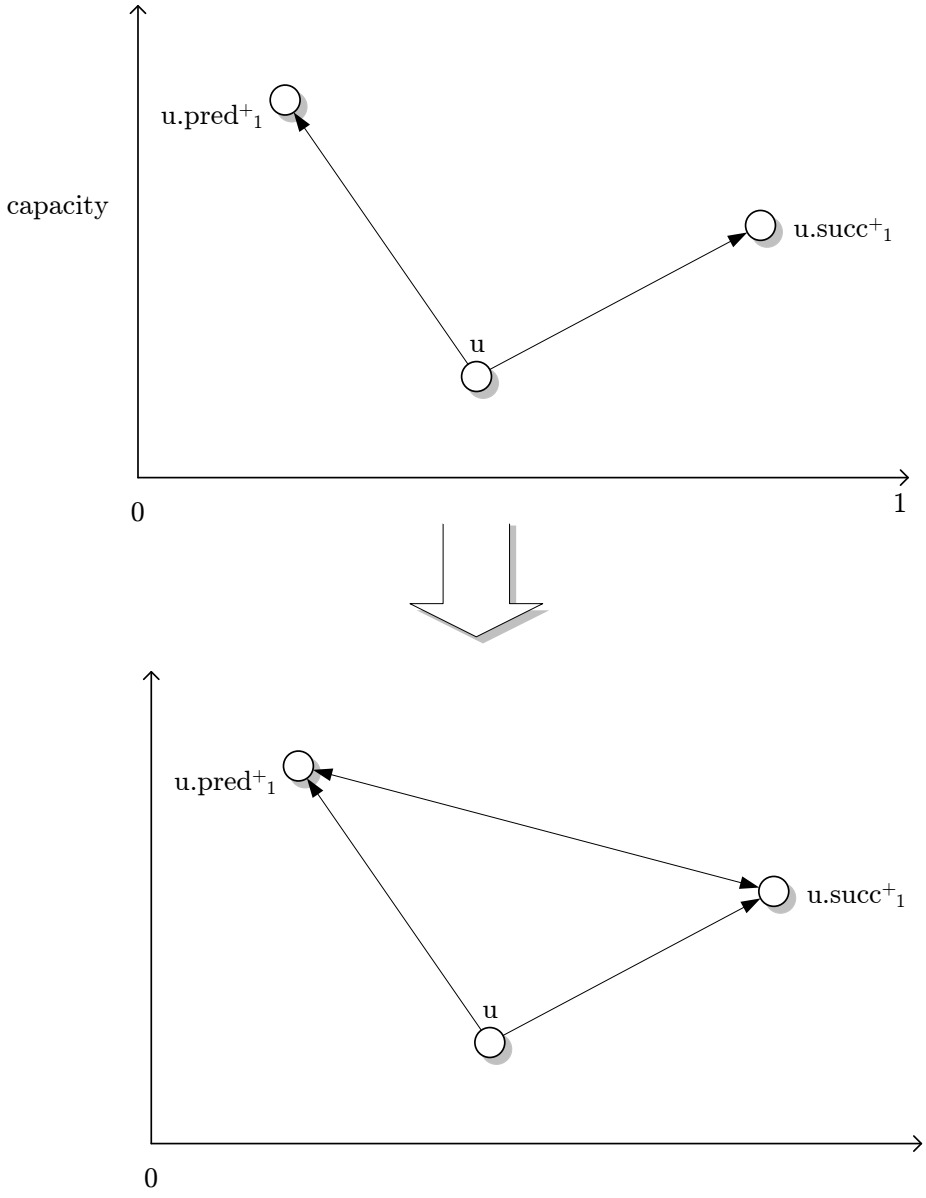


Figure 4.8: Construction of a triangle between u, $u.succ_1^+$, and $u.pred_1^+$

To establish correct P^+ and S^+ lists in each node, a node u sends its $u.P^+$ (resp. $u.S^+$) list periodically to all nodes v in $u.S^-$ (resp. $u.P^-$) by a message $m = (list - update, u.S^+ \cup \{u\})$ (resp. $m = (list - update, u.P^+ \cup \{u\})$) to v.

The last action a node periodically executes is to send a message to each reference in $u.DS$ to check whether u is responsible for the data in the corresponding interval $[a, b]$ by sending a message $m = (check - interval, [a, b], u)$.

If the message predicate is true and u receives a message m, the action u performs depends on the type of the message. If u receives a message $m = (forward, v)$ u checks whether v has to be included in it's internal variables $u.P^+$, $u.S^+$, $u.P^-$ or $u.S^-$. If u doesn't store v, v is delegated. If u receives a message $m = (list - update, list)$, u checks whether the identifiers in $list$ have to be included in it's internal variables $u.P^+$, $u.S^+$, $u.P^-$ or $u.S^-$. If u doesn't store a node v in $list$, v is delegated. If u stores a node v in $u.S^+$ (resp. $u.P^+$) that is not in $list$, v is also delegated as it also has to be in the list of $u.pred_1^+$ (resp. $u.succ_1^+$). The remaining messages are necessary for the data management.

If u receives a message $m = (check - interval, [a, b], v)$ it checks whether v is in $u.S^+$ or $u.P^+$ or has to be included, or delegates v. Then u checks whether $[a, b]$ is in $u.I_u$ and if v is responsible for $[a, b]$. If not, u sends a message $m = (update - interval, IntervalSet)$ to v containing a set of intervals in $[a, b]$ that v is not responsible for and references of the supervising nodes. If u receives a message $m = (update - interval, IntervalSet)$ it forwards all data in intervals in $IntervalSet$ to the corresponding references by a message $m = (forward - data, data)$. If u receives such a message it checks whether the data is in its supervised interval $u.I_u$. If not u forwards the data according to a greedy routing strategy, if u supervises the data it sends a message $m = (store - data, data, u)$ to the responsible node. If u receives such a message it inserts the data, the interval and the corresponding reference in $u.DS$. Note that no identifiers are ever deleted, but always stored or delegated. This ensures the connectivity of the network.

In the following we give a description of the protocol executed by each node in pseudo code in Algorithms 4.3.1- 4.3.9.

4.3.7 Analysis in the ATSS model

In this section we show the correctness of the presented protocol in the ATSS model. We do this by showing that by executing our algorithm any weakly connected network eventually converges to a CONE network and once a CONE network is formed it is maintained in every later state. We further show that in a CONE network the data is stored correctly.

To show convergence we will divide the process of convergence into several phases, such that once one phase is completed its conditions will hold in every later program state.

Theorem 4.3.5 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol P^{CLIQUE} then eventually the graph converges to a graph $G' \in CONE$ (Convergence). If in an initial graph $G \in CONE$ every nodes executes the protocol P^{CONE} then each possible computation leads to a graph $G' \in CONE$ (Closure).*

To prove this theorem we have to introduce some formal definition of the intermediate graphs and some subset of the edges.

Definition 4.3.6 *We define the following subsets of edges:*

Algorithm 4.3.1 P^{CONE}

$m \in u.Ch$ received by u $\rightarrow$ ▷ demand actions depending on the received message
if m=(list-update,List) **then**
  ListUpdate(List)
else if m=(forward,v) **then**
  Forward(v)
else if m=(check-interval,[a,b],v) **then**
  CheckInterval([a,b],v)
else if m=(update-interval,IntervalSet) **then**
  UpdateInterval(IntervalSet)
else if m=(forward-data,data) **then**
  ForwardData(data)
else if m=(store-data,data,interval,v) **then**
  StoreData(data,interval,v)
$u.\tau \rightarrow$ ▷ periodic actions
for all $w \in u.S^-$ **do**
  send m=(list-update,$u.P^+ \cup \{u\}$) to w
for all $w \in u.P^-$ **do**
  send m=(list-update,$u.S^+ \cup \{u\}$) to w
BuildTriangle()
CheckDataIntervals()

Algorithm 4.3.2 BUILDTRIANGLE()

for all $w \in u.S^*$ **do** ▷ periodic introduction of nodes and u itself
  $v^-(w) =$ argmax $\{h(v) : v \in u.S^* \wedge h(v) < h(w)\}$
  send m=(forward,w) to $v^-(w)$ and m'=(forward,$v^-(w)$) to w
for all $w \in u.P^*$ **do**
  $v^+(w) =$ argmin $\{h(v) : v \in u.P^* \wedge h(v) > h(w)\}$
  send m=(forward,w) to $v^+(w)$ and m'=(forward,$v^+(w)$) to w
for all $w \in u.S^- \cup u.P^- \cup \{u.succ_1^+, u.pred_1^+\}$ **do**
  send m=(forward,u) to w
send m=(forward,$u.pred_1^+$) to $u.succ_1^+$ and m'=(forward,$u.succ_1^+$) to $u.pred_1^+$

Algorithm 4.3.3 FORWARD(V)

if $c(v) > c(u) \wedge h(u) < h(v) < h(u.succ_1^+)$ **then** ▷ demand action by a received node id
  send m=(buildtriangle,$u.succ_1^+$) to v
  $u.succ_1^+ = v$
else if $c(v) > c(u) \wedge h(u) > h(v) > h(u.pred_1^+)$ **then**
  send m=(buildtriangle,$u.pred_1^+$) to v
  $u.pred_1^+ = v$
else if $h(v) > h(u)$ **then**
  calculate S_{tmp}^- out of $u.S^-$ and v
  for all $w \in (u.S^- \cup \{v\}) - S_{tmp}^-$ **do**
    $v^-(w) = \text{argmax}\left\{h(v) : v \in S_{tmp}^- \cup \left\{succ_1^+\right\} \wedge h(v) < h(w)\right\}$
    send m=(forward,w) to $v^-(w)$
  $u.S^- = S_{tmp}^-$
else if $h(v) < h(u)$ **then**
  calculate P_{tmp}^- out of $u.P^-$ and v
  for all $w \in (u.P^- \cup \{v\}) - P_{tmp}^-$ **do**
    $v^+(w) = \text{argmin}\left\{h(v) : v \in P_{tmp}^- \cup \left\{pred_1^+\right\} \wedge h(v) > h(w)\right\}$
    send m=(forward,w) to $v^+(w)$
  $u.P^- = P_{tmp}^-$

- $E_{e-connected} = \{(x,y) \in E_e : y \in x.P^+ \cup x.S^+ \cup x.S^- \cup x.P^-\} \subseteq E_e$ *is the subset of explicit edges that will ensure connectivity of the graph.*
- $E_{i-connected} = \{(x,y) \in E_i : \exists m = (forward, y) \in x.Ch\}$ *is the subset of implicit edges that ensure connectivity of the graph.*
- $E_{connected} = E_{e-connected} \cup E_{i-connected}$
- $E_{e-list} = \{(x,y) \in E_e : y = x.P^*[1] \vee y = x.S^*[1]\}$
- $E_{i-list} = \{(x,y) \in E_i : m = (forward, y) \in x.Ch\}$
- $E_{list} = E_{e-list} \cup E_{i-list}$

Definition 4.3.7 *A graph $G = (V,E)$ is a sorted list, if $V = v_0, \cdots v_{n-1}$ with $h(v_i) < h(v_{i+1}) \forall i \in \{0, \cdots n-1\}$ and $E = \{(v_i, v_j) : j = i-1 \vee j = i+1 \; \forall j : 0 < j < n-1\}$.*

Convergence

We start by proving the convergence property of P^{CONE}.

Theorem 4.3.6 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol P^{CONE} then eventually the graph converges to a graph $G' \in CONE$.*

We divide the proof into 3 parts. First we show the preservation of the connectivity of the graph, then we show the convergence to the sorted list and eventually the convergence to the CONE-network.

Algorithm 4.3.4 LISTUPDATE(LIST)

$LList^+ = z \in List : h(z) < h(u) \wedge c(z) > c(u)$ ▷ candidates for $u.P^+$
$LList^- = z \in List : h(z) < h(u) \wedge c(z) < c(u)$ ▷ candidates for $u.P^-$
$RList^+ = z \in List : h(z) > h(u) \wedge c(z) > c(u)$ ▷ candidates for $u.S^+$
$RList^- = z \in List : h(z) > h(u) \wedge c(z) < c(u)$ ▷ candidates for $u.S^-$
calculate P^+_{tmp} out of $u.P^+$ and $LList^+$ ▷ calculate new lists and delegate all nodes not stored in the new lists
$Z = (u.P^+ - LList^+) \cup ((u.P^+ \cup LList^+) - P^+_{tmp})$
if $u.P^+ \neq P^+_{tmp}$ **then**
  for all $z \in Z$ **do**
    send m=(forward,z) to $P^+_{tmp}[1]$
  $u.P^+ = P^+_{tmp}$
calculate S^+_{tmp} out of $u.S^+$ and $RList^+$
$Z = (u.S^+ - RList^+) \cup ((u.S^+ \cup RList^+) - S^+_{tmp})$
if $u.S^+ \neq S^+_{tmp}$ **then**
  for all $z \in Z$ **do**
    send m=(forward,z) to $S^+_{tmp}[1]$
  $u.S^+ = S^+_{tmp}$
calculate P^-_{tmp} out of $u.P^-$ and $LList^-$
for all $w \in (u.P^- \cup LList^-) - P^-_{tmp}$ **do**
  $v^+(w) = \text{argmin}\left\{h(v) : v \in P^-_{tmp} \cup \left\{pred^+_1\right\} \wedge h(v) > h(w)\right\}$
  send m=(forward,w) to $v^+(w)$
$u.P^- = P^-_{tmp}$
calculate S^-_{tmp} out of $u.S^-$ and $RList^-$
for all $w \in (u.S^- \cup RList^-) - S^-_{tmp}$ **do**
  $v^-(w) = \text{argmax}\left\{h(v) : v \in S^-_{tmp} \cup \left\{succ^+_1\right\} \wedge h(v) < h(w)\right\}$
  send m=(forward,w) to $v^-(w)$
$u.S^- = S^-_{tmp}$

Algorithm 4.3.5 CHECKDATAINTERVALS()

for all u.DS[i] **do**
  send m=(check-interval,$[a,b] = u.DS[i]$,u) to $u.DS[i].ref$

Algorithm 4.3.6 CHECKINTERVAL([A,B],V)

if $v \notin u.P^+ \cup u.S^+ \cup u.S^-$ **then**
Forward(v)
IntervalSet := $\emptyset$
i:=1
if $a < h(u)$ **then**
IntervalSet[i]=$[a, h(u)] \cap [a, b]$ ▷ The interval begins left of u, so u can't be the supervising node for the whole interval
IntervalSet[i].ref=$u.P^*[1]$
i:=i+1
if $b > u.S^*[1]$ **then**
IntervalSet[i]=$[u.S^*[1], b] \cap [a, b]$ ▷ The interval ends right of $u.S^*[1]$, so u can't be the supervising node for the whole interval
IntervalSet[i].ref=$u.S^*[1]$
i:=i+1
$[c, d] := I_u(v)$ ▷ $I_u(v)$ is the sub-interval of $u.I_u$ for which v is responsible for
[e,f]:=$([a, b] \cap u.I_u)/I_u(v)$
if $e < c$ **then**
IntervalSet[i]=$[e, c] \cap [a, b]$ ▷ u as the supervising node, knows other nodes responsible for parts of the interval
IntervalSet[i].ref=u
i:=i+1
if $f > d$ **then**
IntervalSet[i]=$[d, f] \cap [a, b]$ ▷ u as the supervising node, knows other nodes responsible for parts of the interval
IntervalSet[i].ref=u
send m=(update-interval,IntervalSet) to v

Algorithm 4.3.7 UPDATEINTERVAL(INTERVALSET)

for all $[a, b] \in IntervalSet$ **do**
for all $[c, d] \in u.DS$ **do**
for all $[e, f] \in \{[c, d] - [a, b]\}$ **do**
l:=$|u.DS|$
u.DS[l+1]=[e,f]
u.DS[l+1].ref=[c,d].ref ▷ references are set to the new supervising node
u.DS:=$u.DS - \{[c, d]\}$
for all $data \in [c, d] \cap [a, b]$ **do**
send m=forward-data(data) to $[a, b].ref$ ▷ data u seems not to be responsible for
delete(data)
Forward([a,b].ref) ▷ references to supervising nodes are forwarded to maintain connectivity
UpdateDS() ▷ Delete all intervals without data, forward the references of the deleted intervals, unite all consecutive intervals with the same reference

Algorithm 4.3.8 FORWARDDATA(DATA)

if $data.id \notin u.I_u$ **then**
  if $data.id \in [u.P^*[1], u]$ **then**
    send m=(forward-data(data)) to $u.P^*[1]$
  else
    send m=(forward-data(data) to
    $w : (h(u) < h(w) < data.id \vee h(u) > h(w) > data.id) \wedge |data.id - h(w)| = \min_{y \in u.P^* \cup u.S^*}\{|data.id - h(y)|\}$,
else
  send m=(store-data,data,$I_u(v)$,u) to $v : data.id \in I_u(v)$

Algorithm 4.3.9 STOREDATA(DATA,INTERVAL,V)

if $\exists i : interval = u.DS[i]$ **then**
  u.DS[i].data := $u.DS[i].data \cup data.id \in u.DS[i]$
  Forward(u.DS[i].ref)
  u.DS[i].ref=v
else
  l:=$|u.DS|$
  u.DS[l+1]=interval
  u.DS[l+1].ref=v
  u.DS[l+1].data=data

In the first part we show that the protocol keeps the network weakly connected and eventually forms a network that is connected by edges $(u, v) \in E_e$ such that $v \in u.P^+ \cup u.S^+ \cup u.S^- \cup u.P^-$ and edges $(u, v) \in E_i$ such that $m = (forward, v) \in u.Ch$.

Lemma 4.3.4 *If u and y are connected in $G^t_{connected}$ at time t then they will be weakly connected at every time $t' > t$.*

Proof. We consider every possible edge (u, v) in $G_{connected}$ and show that u and v stay weakly connected. W.l.o.g. we assume $h(v) > h(u)$.

If $(u, v) \in E^t_{r-connected}$ then there can be the following cases:

- $y \in u.S^+$ at time t, then either v is delegated to $u.succ^+_1$ by a forward message sent to $u.succ^+_1$ and u and v are connected in $G^{t+1}_{connected}$ by $(u, u.succ^+_1) \in E^{t+1}_{e-connected}$ and $(u.succ^+_1, v) \in E^{t+1}_{i-connected}$, or v is stored in $u.S^+$ and $(u, v) \in E^{t+1}_{e-connected}$.
- $v \in u.S^-$ at time t, then either v is delegated to $w^-(v) = \text{argmax}\{h(w) : w \in u.S^* \wedge h(w) < h(v)\}$ or v is also stored in $u.S^-$ at time $t + 1$. By the same arguments as above u and v are connected in $G^{t+1}_{connected}$.

If $(u, v) \in E^t_{i-connected}$ then $m = (forward, v) \in u.Ch$. If u processes m, then either v is stored in $u.P^+, u.S^+, u.S^-, u.P^-$ and $(u, v) \in E^{t+1}_{r-connected}$ or v is delegated to $w^-(v) = \text{argmax}\{h(w) : w \in u.S^* \wedge h(w) < h(v)\}$ and u and v are connected in $G^{t+1}_{connected}$. □

Lemma 4.3.5 *If if there is a state s_t such that G^t is weakly connected then eventually there is a state $s_{t'}$ at time $t' \geq t$ such that $G^{t'}_{connected}$ will be weakly connected.*

Proof. Again we consider every edge (u, v) in G and show that eventually u and v will be connected in $G_{connected}$. Note that we already showed in Lemma 4.3.4, that nodes connected in $G_{connected}$ stay connected in $G_{connected}$. Therefore we only have to consider those edges in $E - E_{connected}$. W.l.o.g. we assume $h(v) > h(u)$.

If $(u, v) \in E^t_e - E^t_{e-connected}$ there can be the following case:

$v \in u.DS[i].ref$ at time t and u has received an interval-update message with a new reference for the data in $u.DS[i]$ or the data is deleted. Then in both cases u delegates v to $w^-(v)$ and u and v are connected in $G_{connected}$. If u does not delegate v, then u eventually sends an check-interval message to v. Then either $u \in v.P^+ \cup v.S^+$ or v delegates u and u and v are weakly connected in $G_{connected}$.

If v is stored in an incoming message in $u.Ch$ then there can be the following cases:

- v is in a list in a list-update message. Then either v is stored in $u.P^+, u.S^+, u.S^-, u.P^-$ or v is delegated to $w^-(v)$ and u and v are weakly connected in $G_{connected}$.
- v is a reference in an interval-update message, then either v is stored as a new reference for some data or if there is no corresponding data v is delegated to $w^-(v)$ and u and v eventually are weakly connected in $G_{connected}$.
- v is a reference in an store-data message, then v is stored as a new reference in $u.DS$. And as already shown u and v are eventually weakly connected.

□

Combining the Lemmas 4.3.4 and 4.3.5 leads to the following theorem:

Theorem 4.3.7 *If G is weakly connected at time t, then at some time $t' \geq t$ $G_{connected}$ will be weakly connected at every time $t'' \geq t'$.*

In the next part we show that if $G_{connected}$ is weakly connected eventually a sorted list will be formed.

Theorem 4.3.8 *If $G_{connected}$ is weakly connected at a state s_t then eventually $G^{t'}_{e-list}$ will be a sorted list in a state $s_{t'}$ $t' \geq t$ and stay a sorted list in every later state.*

Before we show the theorem we show some helping lemmas.

Lemma 4.3.6 *Eventually all nodes $u.S^*[i]$ and $u.S^*[i+1]$ (resp. $u.P^*[i]$ and $u.P^*[i+1]$) with $i < |u.S^*|$ will be connected and stay connected in every state after over nodes w with $h(u.S^*[i]) < h(w) < h(u.S^*[i+1])$.*

Proof. In the periodic action u executes $Build - Triangle()$, in which u introduces every pair of nodes $u.S^*[i], u.S^*[i+1]$ to each other (see Figure 4.9). The connecting path for such a pair only changes if w.l.o.g. $u.S^*[i]$ delegates $u.S^*[i+1]$, but then $u.S^*[i]$ can delegate $u.S^*[i+1]$ only to a node v with $h(u.S^*[i]) < h(v) < h(u.S^*[i+1])$. By using this argument inductively $u.S^*[i]$ and $u.S^*[i+1]$ stay connected in every state after over nodes w with $h(u.S^*[i]) < h(w) < h(u.S^*[i+1])$. □

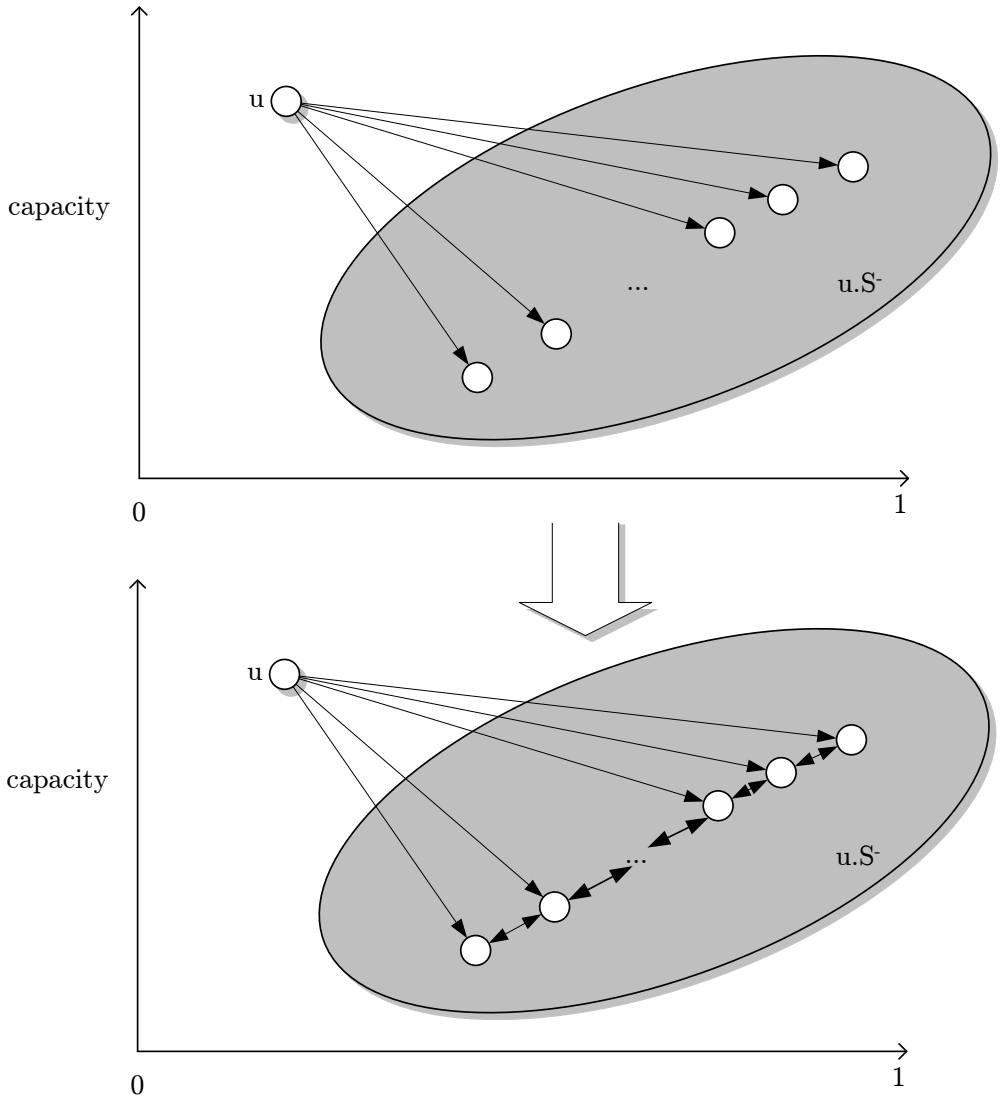


Figure 4.9: A node u introduces the nodes in $u.S^-$ to each other in Build-Triangle()

Lemma 4.3.7 *If* $(u,v) \in E_{connected}$ *and* $(u,z) \in E^t_{connected}$ *and* $h(u) < h(v) < h(z)$ *(resp.* $h(u) > h(v) > h(z)$*) then eventually* $(v',z) \in E^{t'}_{connected}$ *with* $h(u) < h(v') < h(z)$ *and* $t' \geq t$ *(resp.* $h(u) > h(v') > h(z)$*) and* u *and* z *are connected over nodes* $w : h(u) < h(w) \leq h(v')$ *(resp.* $w : h(u) > h(w) \geq h(v')$ *and in particular over* $u.S^*[1]$ *(resp.* $u.P^*[1]$*).*

Proof. W.l.o.g we assume $h(u) < h(v) < h(z)$. If u delegates z to a node v' then obviously $h(u) < h(v') < h(z)$ and u and z are connected over $w = v'$. In case z is not delegated z is stored in $u.S^+$ or $u.S^-$ or in a message $m = (forward, z) \in u.Ch$.

If z is stored in $u.S^-$ and $(u,v) \in E_{connected}$ and $h(u) < h(v) < h(z)$ then either $v \in u.S^-$ or $m' = (forward, v) \in u.Ch$. Eventually m' is processed by u and either v is delegated, then there is another node $w' \in u.S^- : h(u) < h(w') < h(v) < h(z)$ or v is stored in $u.S^-$. Thus eventually $z \in u.S^-$ and another node $w' \in u.S^-$ such that $h(u) < h(w') < h(z)$. From all such nodes $w' \in u.S^-$ such that $h(u) < h(w') < h(z)$ u introduces z to $v' = \text{argmax}\{h(x) : x \in u.S^- \wedge h(x) < h(z)\}$ and $(v',z) \in E_{connected}$ with $h(u) < h(v') < h(z)$ and by the same arguments as above u and z stay connected over nodes $h(u) < h(w) \leq h(v')$.

If $z \in u.S^+$ and $z = u.succ^+_1$ the same analysis as for $z \in u.S^-$ can be applied. If $z \in u.S^+$ and $z \neq u.succ^+_1$ eventually u will receive the $v'.S^+$ list of $v' = u.succ^+_1$. If $z \in v'.S^+$ then $(v',z) \in E^{connected}$ with $h(u) < h(v') < h(z)$ and by the same arguments as above u and z stay connected over nodes $h(u) < h(w) < h(v')$. Otherwise u sends a message $m = (forward, z)$ to v' and again $(v',z) \in E_{connected}$ with $h(u) < h(v') < h(z)$ and by the same arguments as above u and z stay connected over nodes $h(u) < h(w) < h(v')$.

If $m = (forward, z) \in u.Ch$, then eventually u processes m and either stores z in $u.S^+$ or $u.S^-$ and we can apply one of the cases above or z is delegated.

In all cases $v' \in u.S^*$ and thus u and z are also connected over $u.S^*[1]$, as all nodes in $u.S^*$ are connected according to Lemma 4.3.6. □

Lemma 4.3.8 *If* $(u,z) \in E^t_{connected}$ *with* $h(u) < h(z)$ *(resp.* $h(u) > h(z)$*) then eventually* $(z,v) \in E^{t'}_{connected}$ $t' \geq t$ *with* $h(u) < h(v) < h(z)$ *(resp.* $h(u) > h(v) > h(z)$*)and* u *and* v *are connected over nodes* w *with* $h(u) < h(w) < h(v)$ *(resp.* $h(u) > h(w) > h(v)$*).*

Proof. W.l.o.g we assume $h(u) < h(z)$. If $(u,z) \in E^t_{connected}$ and $(u,v) \in E^t_{connected}$ and $h(u) < h(v) < h(z)$ we can apply Lemma 4.3.7 and eventually $(v',z) \in E^{t'}_{connected}$ with $h(u) < h(v') < h(z)$ and u and v' are connected over nodes w $h(u) < h(w) < h(v')$ in every state after.

Now if $(v',y) \in E^{t'}_{connected}$ with $h(v') < h(y) < h(z)$ we might again apply the lemma. Obviously we only can apply Lemma 4.3.7 a finite number of times until there is a node w' such that $(w',z) \in E^{t''}_{connected}$ and there is no $(w',y') \in E^{t''}_{connected}$ with $h(w') < h(y') < h(z)$. Then still u and w' are connected over nodes $h(u) < h(w) < h(w')$. Then either $z = w'.S^*[1]$ or $m = (forward, z) \in w'.Ch$. If $z = w'.S^*[1]$ then eventually w' will introduce itself to z by a message $m' = (forward, w')$, then $(z,w') \in E_{connected}$ with $h(u) < h(w') < h(z)$ and u and w' are connected over nodes $w : h(u) < h(w) < h(w')$. Otherwise as soon as w' processes m' $w'.S^*[1]$ is set to z and the same arguments as in the first case hold. □

Before we prove the theorem we introduce some additional definitions.

Definition 4.3.8 *We define an* undirected path p *as a sequence of edges* $(v_0,v_1),(v_1,v_2),\cdots,(v_{k-1},v_k)$*, such that* $\forall i \in \{1,\cdots,k\} : (v_i,v_{i-1}) \in E_{connected} \vee (v_{i-1},v_i) \in E_{connected}$*. We further define*

$v_{min} = argmin\{h(v) : v \in p\}$ *and* $v_{max} = argmax\{h(v) : v \in p\}$. *Then the* range of a path $range(p)$ *is given by* $range(p) = v_{max} - v_{min}$.

Now we are ready to prove Theorem 4.3.8.

Proof. Let v_i and v_{i+1} be a pair of nodes connected in the sorted list. Then as $G_{connected}$ is weakly connected there is an undirected path connecting v_i and v_{i+1}. Let p^t be such a path at time t. We show that there is a path $p^{t'}$ with $t' \geq t$ that connects v_i and v_{i+1} weakly such that $range(p^t) > range(p^{t'})$. Let u^t_{min} and u^t_{max} be the smallest and greatest node on the path p^t that define the range of p^t. W.l.o.g. we show that the range decreases as $h(u^t_{min})$ increases for larger t.

The node u^t_{min} is connected to nodes w_1 and w_2 on p^t. If $(w_1, u^t_{min}) \in E^t_{connected}$ and $(u^t_{min}, w_1) \notin E^{t+1}_{connected}$, then according to Lemma 4.3.8 eventually $(u^t_{min}, v_1) \in E_{connected}$ and w_1 and v_1 are connected over nodes w' such that $h(v_1) < h(w') < h(w_1)$. The same holds for w_2. Thus eventually $(u^t_{min}, v_1) \in E^t_{connected}$ and $(u^t_{min}, v_2) \in E^t_{connected}$ and w_1 and v_1 are connected over nodes w' with $h(v_1) < h(w') < h(w_1)$ and w_2 and v_2 are connected over nodes $w'' : h(v_2) < h(w'') < h(w_2)$. Then either $v_1 = v_2$ and we can construct another path connecting u and y over w_1 and w_2 with $u^{t'}_{min} = v_1 = v_2$, and $h(u^{t'}_{min}) > h(u^{t'}_{min})$, otherwise $h(v_1) < h(v_2)$ or $h(v_2) < h(v_1)$. W.l.o.g. we assume $h(v_1) < h(v_2)$. Then according to Lemma 4.3.7 eventually $(v'_2, v_2) \in E^{t'}_{connected}$ and v'_2 and u^t_{min} are connected over nodes $w'' : h(u^t_{min}) < h(w'') < h(v'_2)$. Either $v_1 = u^t_{min}.S^*[1]$ or also according to Lemma 4.3.7 $(v'_1, v_1) \in E_{connected}$ and v'_1 and u^t_{min} are connected over nodes $w'' : h(u^{min}) < h(w'')' < h(v'_1)$. Note that according to the proof of Lemma 4.3.7 v'_1 and v'_2 have to be in $u^{min}.S^*$ at the time t' the edge $(v'_2, v_2) \in E^{t'}_{connected}$ resp. $(v'_1, v_1) \in E^{t'}_{connected}$ is created. Then according to Lemma 4.3.6 v'_1 and v'_2 are also connected to $u^{min}.S^*[1]$. Thus again we can construct another path connecting v_i and v_{i+1} over w_1 and w_2 with $u^{t'}_{min} = u^t_{min}.S^*[1] \wedge h(u^t_{min}.S^*[1]) > h(u^t_{min})$.

Thus eventually a connecting path can be found with a strict smaller range and by applying these arguments a finite number of times $(v_i, v_{i+1}) \in E^{t*}_{connected}$ and $(v_{i+1}, v_i) \in E^{t*}_{connected}$ in some state s_{t*} at time t^*. Then if $v_{i+1} \in v_i.S^*$ $v_i.S^*[1] = v_{i+1}$ otherwise $m = (forward, v_{i+1}) \in v_i.Ch$ will eventually be processed and $v_i.S^*[1]$ is set to v_{i+1}. By the same arguments eventually $v_i = v_{i+1}.P^*[1]$. As this holds for every pair v_i, v_{i+1} of consecutive nodes in the sorted list, eventually G_{e-list} is a sorted list and stays a sorted list in every later state. □

In the last part we show that once the network has stabilized into a sorted list, it eventually also stabilizes into the legal CONE-network.

Theorem 4.3.9 *If* G^t_{e-list} *is a sorted list then eventually the computation of* P^{CONE} *reaches a state* $s_{t'}$ *such that* $G^{t'} \in CONE$.

We will first prove that eventually for every node u, $u.succ^+_1 = succ^+_1(u)$ and $u.pred^+_1 = pred^+_1(u)$.

Lemma 4.3.9 *If* G^t_{e-list} *is a sorted list eventually for every node* u $u.succ^+_1 = succ^+_1(u)$ *and* $u.pred^+_1 = pred^+_1(u)$.

Proof. According to Theorem 4.3.8 the computation reaches a state s_{t_0} such that $G^{t_0}_{e-list}$ is a sorted list. We then prove the lemma by induction on the capacities of the nodes. In fact we show the following hypothesis: If for all nodes v with $c(v) < c(u)$ $v.succ^+_1 = succ^+_1(v)$ and $v.pred^+_1 = pred^+_1(v)$ then eventually $u.succ^+_1 = succ^+_1(u)$ and $u.pred^+_1 = pred^+_1(u)$.

We start the induction on the node v_{min} with the smallest capacity. Obviously for this node $succ(v_{min}) = succ_1^+(v_{min})$ and $pred(v_{min}) = pred_1^+(v_{min})$. As G^t_{e-list} is a sorted list also $v_{min}.S^*[1] = succ(v_{min})$ and $v_{min}.P^*[1] = pred(v_{min})$. Thus v_{min} is already connected to $succ_1^+(v_{min})$ and $pred_1^+(v_{min})$, then $v_{min}.succ_1^+ = succ_1^+(v_{min})$ and $v_{min}.pred_1^+ = pred_1^+(v_{min})$.

In the inductive step let u be a node for which for all nodes v with $c(v) < c(u)$ $v.succ_1^+ = succ_1^+(v)$ and $v.pred_1^+ = pred_1^+(v)$. Let $z = succ_1^+(u)$. Then let $w_1, w_2 \cdots w_l$ be the nodes in the sorted list between u and z. We know that for each w_i $c(w_i) < c(u)$ and thus $w_i.succ_1^+ = succ_1^+(w_i)$ and $w_i.pred_1^+ = pred_1^+(w_i)$. Let $w_j = \text{argmax}\{c(w_i) : i \leq l\}$. Then for w_j $succ_1^+(w_j) = z$ and $pred_1^+(w_i) = u$.

Thus eventually w_j will introduce z to u and vice versa in the Build-Triangle() action and $u.succ_1^+ = succ_1^+(u) = z$. By symmetric arguments we can show that also $u.pred_1^+ = pred_1^+(u)$. □

Lemma 4.3.10 *If G^t_{e-list} is a sorted list and for every node u $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$ then eventually $u.S^+ = S^+(u)$ and $u.P^+ = P^+(u)$ and $u.S^- = S^-(u)$ and $u.P^- = P^-(u)$.*

Proof. We begin by showing that $u.S^- = S^-(u)$ and $u.P^- = P^-(u)$. If G^t_{e-list} is a sorted list and for every node u $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$ then eventually each node u will receive a message $m = (build - triangle, v)$ from each node v with $v.succ_1^+ = succ_1^+(v) = u$ or $v.pred_1^+ = pred_1^+(v) = u$. Then $u.S^- = S^-(u)$ and $u.P^- = P^-(u)$. We show that eventually $u.S^+ = S^+(u)$ and $u.P^+ = P^+(u)$ by induction on the capacity of the nodes. We prove the following hypothesis: If for all nodes v with $c(v) > c(u)$ $v.S^+ = S^+(v)$ and $v.P^+ = P^+(v)$ then eventually $u.S^+ = S^+(u)$ and $u.P^+ = P^+(u)$. We start the induction on the node $u = v_{max}$ with the greatest capacity. Obviously for v_{max} $v_{max}.S^+ = S^+(v_{max})$ and $v_{max}.P^+ = P^+(v_{max})$.

In the inductive step let u be a node for which for all nodes v with $c(v) > c(u)$ $v.S^+ = S^+(v)$ and $v.P^+ = P^+(v)$. Then we know by the first part of the proof that $u \in u.succ_1^+.P^-$ and thus $u.succ_1^+$ sends a message $m = (list - update, u.succ_1^+.S^+ \cup \{u.succ_1^+\})$. As $u.succ_1^+.S^+ = S^+(succ_1^+(u))$ also $u.S^+ = S^+(u)$. By symmetric arguments we can show that $u.P^+ = P^+(u)$. □

Combing Lemma 4.3.9 and Lemma 4.3.10 Theorem 4.3.9 immediately follows.

Combining Theorem 4.3.7, Theorem 4.3.8 and Theorem 4.3.9 we can show that Theorem 4.3.6 holds, and by our protocol each weakly connected network converges to a CONE network.

Closure and Correctness of the data structure

We showed that from any initial state we eventually reach a state in which the network forms a correct CONE network. We now need to show that in this state the explicit edges remain stable and also that each node stores the data it is responsible for.

Theorem 4.3.10 *If in an initial graph $G \in CONE$ every nodes executes the protocol P^{CONE} then each possible computation leads to a graph $G' \in CONE$.*

Proof. The graph $G_e = (V, E_e)$ only changes if the explicit edge set is changed. So if we assume that $G \in CONE$ at time t and for $t' > t$ $G' \notin CONE$ then we added or deleted at least one explicit edge. Let $(u, v) \in E_e^t$ at time t. W.l.o.g. we assume $h(u) < h(v)$. Either $v \in S^+(u)$ or $v \in S^-(u)$. In both cases the edge is only deleted if u knows a node w with $h(u) < h(w) < h(v)$ and $c(w) > c(v)$. As following from Theorem 4.3.9 all internal neighborhoods are correct in G there can not be such a node w. By the same argument also no new edges are created. Thus $G' \in CONE$ at time t'. □

So far we have shown that by our protocol eventually a CONE network is formed. It remains to show that also by our protocol eventually each node stores the data it is responsible for.

Theorem 4.3.11 *If $G \in CONE$ eventually each node stores exactly the data it is responsible for.*

Proof. According to Theorem 4.3.2 each node knows which node is responsible for parts of the interval it supervises. In our described algorithm each node u checks whether it is responsible for the data it currently stores by sending a message to the node v that u assumes to be supervising the corresponding interval. If v is supervising the interval and u is responsible for the data, then u simply keeps the data. If v is not supervising the data or u is not responsible for the data then v sends a reference to u with the id of anode that v assumes to be supervising the interval. Then u forwards the data to the new reference and does not store the data. By forwarding the data by Greedy Routing it eventually reaches a node supervising the corresponding interval, this node then tells the responsible node to store the data. Thus eventually all data is stored by nodes that are responsible for the data. □

4.3.8 Protocol $P^{CONEsync}$

In this section we give a description of the the distributed algorithm for the synchronous setting. We modify the protocol P^{CONE} in a way that a node u that receives list-update messages from nodes w that are not $u.pre_1^+$ or $u.succ_1^+$ at most once, i.e. u sends a message back containing the id of a node v with $h(v)$ between $h(u)$ and $h(w)$ and $c(u) < c(v) < c(w)$. Then in the following rounds w can only send list-update messages to v, but not to u. We therefore collect all sending nodes of a list-update message in a set LS and all received lists in a set $LIST$. Furthermore we substitute the forward messages sent periodically by a node to introduce itself by introduction messages. If a node u receives an introduction message by a node w not in $u.S^*$ or $u.P^*$, u informs the sender of the message about a node v it should send the message to instead of u. Thus u receives introduction messages by nodes not in $u.S^*$ or $u.P^*$ at most once. Otherwise introduction messages are simply treated as forward messages. All introduction and forward messages are collected in a set $FORWARD$, which is then sorted. Then nodes are only forwarded to preceding nodes in the sorted set. In the following we give a description of the protocol executed by each node in pseudo code in Algorithms 4.3.10- 4.3.18.

4.3.9 Analysis in the STSS model

In this section we analyze $P^{CONEsync}$ according to the STSS model. In particular we consider the stabilization time, stabilization work and the maintenance work. Remind that D is the number of data items stored in the system and that r is the number of nodes one node keeps references to.

Theorem 4.3.12 *If an initial graph $G \in IT$ is weakly connected and each node executes the protocol $P^{CONEsync}$ then the graph converges to a graph $G' \in CONE$ (Convergence) with a stabilization time of $\mathcal{O}(n)$ and a stabilization work of $\mathcal{O}(n^2 + Dn \log n)$. If in an initial graph $G \in CONE^*$ every node executes the protocol $P^{RE-CHORD}$ then each possible computation leads to a graph $G' \in CONE$ (Closure) with a maintenance work of $\mathcal{O}(r + \log^2 n)$. In a legal state it takes $\mathcal{O}(\log n)$ rounds and $\mathcal{O}(\log^3 n)$ messages w.h.p. to recover and stabilize after a new real node joins or leaves the network.*

Algorithm 4.3.10 $P^{CONEsync}$

message $m \in u.Ch \rightarrow$
$FORWARD = \emptyset$ ▷ Ids of nodes that are received by forward messages
$LIST = \emptyset$ ▷ IDs received by list-update messages
$LS = \emptyset$ ▷ IDs of senders of list-update messages
$CS = \emptyset$ ▷ IDs of senders of check-interval messages and received intervals
while $m \in u.Ch$ **do** ▷ demand actions depending on the received message
  if m=(list-update,List,v) **then**
    $LIST = LIST \cup List$
    $LS = LS \cup \{v\}$
  else if m=(forward,v) **then**
    $FORWARD = FORWARD \cup \{v\}$
  else if m=(introduction,v) **then**
    $FORWARD = FORWARD \cup \{v\}$
  else if m=(check-interval,[a,b],v) **then**
    $CS = CS \cup \{([a,b],v)\}$
  else if m=(update-interval,IntervalSet) **then**
    UpdateInterval(IntervalSet)
  else if m=(forward-data,data) **then**
    ForwardData(data)
  else if m=(store-data,data,interval,v) **then**
    StoreData(data,interval,v)
ListUpdate()
CheckInterval(CS)
Forward(FORWARD)
$u.\tau \rightarrow$ ▷ periodic actions
for all $w \in u.S^-$ **do**
  send m=(list-update,$u.P^+ \cup \{u\}, u$) to w
for all $w \in u.P^-$ **do**
  send m=(list-update,$u.S^+ \cup \{u\}, u$) to w
BuildTriangle()
CheckDataIntervals()

Algorithm 4.3.11 BUILDTRIANGLE()

for all $w \in u.S^*$ **do** ▷ periodic introduction of nodes and u itself
  $v^-(w) = \text{argmax}\{h(v) : v \in u.S^* \wedge h(v) < h(w)\}$
  send m=(forward,w) to $v^-(w)$ and m'=(forward,$v^-(w)$) to w
for all $w \in u.P^*$ **do**
  $v^+(w) = \text{argmin}\{h(v) : v \in u.P^* \wedge h(v) > h(w)\}$
  send m=(forward,w) to $v^+(w)$ and m'=(forward,$v^+(w)$) to w
for all $w \in u.S^* \cup u.P^*$ **do**
  send m=(introduction,u) to w
send m=(forward,$u.pred_1^+$) to $u.succ_1^+$ and m'=(forward,$u.succ_1^+$) to $u.pred_1^+$

Algorithm 4.3.12 FORWARD(FORWARD)

Calculate P^*_{new}, S^*_{new} (including pre^+_{1new}, $succ^+_{1new}$) out of $FORWARD \cup u.S^* \cup u.P^*$
$W = (FORWARD \cup u.S^* \cup u.P^*) - (P^*_{new} \cup S^*_{new})$
$W' = FORWARD \cup u.S^* \cup u.P^*$
for all $w \in W$ **do**
  if $h(w) < h(u)$ **then**
    $v^+(w) =$ argmin $\{h(v) : h(v) > h(w) \wedge v \in W'\}$
    send message $m = (forward, w)$ to $v^+(w)$
    if w is received by an introduction message **then**
      $v^+(w) =$ argmin $\{h(v) : h(v) > h(w) \wedge c(v) > c(w) \wedge v \in W'\}$
      send message $m = (forward, v^+(w))$ to w
  else
    $v^-(w) =$ argmax $\{h(v) : h(v) < h(w) \wedge v \in W'\}$
    send message $m = (forward, w)$ to $v^-(w)$
    if w is received by an introduction message **then**
      $v^-(w) =$ argmax $\{h(v) : h(v) < h(w) \wedge c(v) > c(w) \wedge v \in W'\}$
      send message $m = (forward, v^-(w))$ to w
$u.P^* = P^*_{new}$
$u.S^* = S^*_{new}$

Stabilization Time

We start by showing the stabilization time of the protocol $P^{CONEsync}$. Note that in the $STSS$ model we execute all enabled actions in one round, i.e. all messages in $u.Ch$ are received by a node u. Thus we slightly adapt the notation of E^t and E^t_e. Instead of considering the changed edge sets after each single action we consider the changed edge sets after executing all enabled actions for all nodes in one round.

Theorem 4.3.13 *If an initial graph $G \in IT$ is weakly connected and each nodes executes the protocol $P^{CONEsync}$ then the graph converges to a graph $G' \in CONE$ (Convergence) with a stabilization time of $\mathcal{O}(n)$.*

To prove this theorem we use the same formal definitions of the intermediate graphs and subsets of the edges like in the analysis in the ATSS model.

Like in the analysis of P^{CONE} in the ATSS model we will prove the theorem by showing some intermediate steps. We start by showing that in $\mathcal{O}(n)$ rounds all nodes are weakly connected in E_{list} and stay weakly connected. In the following analysis we treat every introduction messages like a simple forward message, as for received introduction messages only $\mathcal{O}(1)$ additional forward messages are sent, i.e. every argument used for a forward message also holds for introduction messages.

Theorem 4.3.14 *If $G_{connected}$ is weakly connected at a state s_t then after at most $\mathcal{O}(n)$ rounds $G^{t'}_{e-list}$ will be a sorted list in a state $s_{t'}$ and stay a sorted list in every later state.*

In order to show the theorem we first show some helpful lemmas.

Lemma 4.3.11 *If u and v are weakly connected in $G^t_{connected}$ at time t then they will be weakly connected at every time $t' > t$.*

Algorithm 4.3.13 LISTUPDATE(LIST)

$LList^+ = z \in List : h(z) < h(u) \wedge c(z) > c(u)$ ▷ candidates for $u.P^+$
$LList^- = z \in List : h(z) < h(u) \wedge c(z) < c(u)$ ▷ candidates for $u.P^-$
$RList^+ = z \in List : h(z) > h(u) \wedge c(z) > c(u)$ ▷ candidates for $u.S^+$
$RList^- = z \in List : h(z) > h(u) \wedge c(z) < c(u)$ ▷ candidates for $u.S^-$
calculate P^+_{new} out of $u.P^+$ and $LList^+$ ▷ calculate new lists and delegate all nodes not stored in the new lists
calculate S^+_{new} out of $u.S^+$ and $RList^+$
calculate P^-_{new} out of $u.P^-$ and $LList^-$
calculate S^-_{new} out of $u.S^-$ and $RList^-$
$N = List \cup u.S^+ \cup u.S^- \cup u.P^+ \cup u.P^-$
$L = S^+_{new} \cup S^-_{new} \cup P^+_{new} \cup P^-_{new}$
for all $w \in N - S$ **do** ▷ Nodes are delegated
  if $h(w) < h(u)$ **then**
    $v^+(w) =$ argmin $\{h(v) : h(v) > h(w) \wedge v \in N\}$
    send message $m = (forward, w)$ to $v^+(w)$
    if w is a sender of a list-update message **then** ▷ i.e. $w \in LS$
      $v^+(w) =$ argmin $\{h(v) : h(v) > h(w) \wedge c(v) > c(w) \wedge v \in N\}$
      send message $m = (forward, v^+(w))$ to w
  else
    $v^-(w) =$ argmax $\{h(v) : h(v) < h(w) \wedge v \in N\}$
    send message $m = (forward, w)$ to $v^-(w)$
    if w is a sender of a list-update message **then**
      $v^-(w) =$ argmin $\{h(v) : h(v) < h(w) \wedge c(v) > c(w) \wedge v \in N\}$
      send message $m = (forward, v^-(w))$ to w
for all $w \in S : w$ *is a sender of a list-update message* **do**
  if $w \neq P^+_{new}[1] \wedge w \neq S^+_{new}[1]$ **then** ▷ w is not the new $u.pre^+_1$ or $u.succ^+_1$
    if $h(w) < h(u)$ **then**
      $v^+(w) =$ argmin $\{h(v) : h(v) > h(w) \wedge v \in S\}$
      send message $m = (forward, v^+(w))$ to w
    else
      $v^-(w) =$ argmin $\{h(v) : h(v) < h(w) \wedge v \in S\}$
      send message $m = (forward, v^-(w))$ to w
$u.P^+ = P^+_{new}$
$u.S^+ = S^+_{new}$
$u.P^- = P^-_{new}$
$u.S^- = S^-_{new}$

Algorithm 4.3.14 CHECKDATAINTERVALS()

for all u.DS[i] **do**
  send m=(check-interval,$[a, b] = u.DS[i]$,u) to $u.DS[i].ref$

Algorithm 4.3.15 CHECKINTERVAL(CS

for all $([a,b],v) \in CS$ **do**
  if $v \notin u.P^+ \cup u.S^+ \cup u.S^-$ **then**
    $FORWARD = FORWARD \cup \{v\}$
  IntervalSet := $\emptyset$
  i:=1
  if $a < h(u)$ **then**
    IntervalSet[i]=$[a,h(u)] \cap [a,b]$ ▷ The interval begins left of u, so u can't be the supervising node for the whole interval
    IntervalSet[i].ref=$u.P^*[1]$
    i:=i+1
  if $b > u.S^*[1]$ **then**
    IntervalSet[i]=$[u.S^*[1],b] \cap [a,b]$ ▷ The interval ends right of $u.S^*[1]$, so u can't be the supervising node for the whole interval
    IntervalSet[i].ref=$u.S^*[1]$
    i:=i+1
  $[c,d] := I_u(v)$ ▷ $I_u(v)$ is the sub-interval of $u.I_u$ for which v is responsible for
  [e,f]:=$([a,b] \cap u.I_u)/I_u(v)$
  if $e < c$ **then**
    IntervalSet[i]=$[e,c] \cap [a,b]$ ▷ u as the supervising node, knows other nodes responsible for parts of the interval
    IntervalSet[i].ref=u
    i:=i+1
  if $f > d$ **then**
    IntervalSet[i]=$[d,f] \cap [a,b]$ ▷ u as the supervising node, knows other nodes responsible for parts of the interval
    IntervalSet[i].ref=u
  send m=(update-interval,IntervalSet) to v

Algorithm 4.3.16 UPDATEINTERVAL(INTERVALSET)

for all $[a,b] \in IntervalSet$ **do**
  for all $[c,d] \in u.DS$ **do**
    for all $[e,f] \in \{[c,d] - [a,b]\}$ **do**
      l:=$|u.DS|$
      u.DS[l+1]=[e,f]
      u.DS[l+1].ref=[c,d].ref ▷ references are set to the new supervising node
      u.DS:=$u.DS - \{[c,d]\}$
    for all $data \in [c,d] \cap [a,b]$ **do**
      send m=forward-data(data) to $[a,b].ref$ ▷ data u seems not to be responsible for delete(data)
      $FORWARD = FORWARD \cup \{[a,b].ref\}$ ▷ references supervising nodes are forwarded to maintain connectivity
  UpdateDS() ▷ Delete all intervals without data, forward the references of the deleted intervals, unite all consecutive intervals with the same reference

Algorithm 4.3.17 FORWARDDATA(DATA)

if $data.id \notin u.I_u$ **then**
  if $data.id \in [u.P^*[1]], u]$ **then**
    send m=(forward-data(data)) to $u.P^*[1]$
  else
    send m=(forward-data(data) to
    $w : (h(u) < h(w) < data.id \vee h(u) > h(w) > data.id) \wedge |data.id - h(w)| = \min_{y \in u.P^* \cup u.S^*}\{|data.id - h(y)|\}$,
else
  send m=(store-data,data,$I_u(v)$,u) to $v : data.id \in I_u(v)$

Algorithm 4.3.18 STOREDATA(DATA,INTERVAL,V)

if $\exists i : interval = u.DS[i]$ **then**
  u.DS[i].data := $u.DS[i].data \cup data.id \in u.DS[i]$
  $FORWARD = FORWARD \cup \{u.DS[i].ref\}$
  u.DS[i].ref=v
else
  l:=$|u.DS|$
  u.DS[l+1]=interval
  u.DS[l+1].ref=v
  u.DS[l+1].data=data

Proof. We show that for each existing edge $(u, v) \in E^t_{connected}$ either the edge remains and $(u, v) \in E^{t+1}_{connected}$ or a path connecting u, v exists. Obviously u and v stay weakly connected as long as an edge $(u, v) \in E^{t+1}_{connected}$ exists. We therefore assume $(u, v) \in E^t_{connected}$ and $(u, v) \notin E^{t+1}_{connected}$. If v is stored in an internal variable of u then there can be the following cases:

- $v \in u.P^+ \cup u.S^+$ at time t, then v is delegated and u and v stay connected over the edges $(u, u.pred^+_1) \in E^{t+1}_{e-connected}$ and a path of implicit edges $(u.pred^+_1, w_1), (w_1, w_2) \cdots (w_i, v) \in E^{t+1}_{i-connected}$.

- $v \in u.S^- \cup u.P^-$ at time t, then v is delegated and u and v stay connected in $E^{t+1}_{connected}$ by a path of implicit edges $(w_1, w_2), \cdots (w_i, v) \in E^{t+1}_{i-connected}$. and one explicit edge $(u, z) \in E^{t+1}_{e-connected}$ with $z \in u.S^- \cup u.P^-$ at time $t + 1$.

- $v \in u.DS[i].ref$ at time t then u has received an interval-update message with a new reference for the data in $u.DS[i]$ or the data is deleted. Then in both cases u delegates v to $w^-(v)$ (resp. $w^+(v)$) and u and v stay connected in $E^{t+1}_{connected}$.

If v is stored in an incoming message m in $u.Ch$, then then there can be the following cases when m is received:

- v is in a list in a list-update message. Then either v is stored in $u.P^+, u.S^+, u.S^-, u.P^-$ or v is delegated and u and v stay connected as described in the case of $v \in u.P^+ \cup u.S^+$.

- v is the node sending a check-interval message. Then either v is stored in $u.P^+, u.S^+$ or delegated to $w^-(v)$ (resp. $w^+(v)$) and u and v stay connected.
- v is a reference in an interval-update message, then either v is stored as a new reference for some data or if there is no corresponding data v is delegated to $w^-(v)$ (resp. $w^+(v)$) and u and v stay connected.
- v is a reference in an store-data message, then v is stored as a new reference in $u.DS$.
- v is an identifier in a forward message, then v is either stored in $u.P^+, u.S^+, u.S^-, u.P^-$ or v is delegated to $w^-(v)$ (resp. $w^+(v)$) and u and v stay connected.

□

Lemma 4.3.12 *If if there is a state s_t such that G^t is weakly connected then after $\mathcal{O}(1)$ rounds the computation reaches a state $s_{t'}$ $t' \geq t$ such that $G^{t'}_{connected}$ is weakly connected.*

Proof. Again we consider every edge (u, v) in G and show that eventually u and v will be connected in $G_{connected}$. Note that we already showed in Lemma 4.3.11, that nodes connected in $G_{connected}$ stay connected in $G_{connected}$. Therefore we only have to consider those edges in $E - E_{connected}$.

If $(u, v) \in E^t_e - E^t_{e-connected}$ there can be the following case:

Either $v \in u.DS[i].ref$ at time t and u has received an interval-update message with a new reference for the data in $u.DS[i]$ or the data is deleted. Then in both cases u delegates v to $w^-(v) = \text{argmax}\,\{h(w) : w \in u.S^* \wedge h(w) < h(v)\}$ (resp. $w^+(v)$) and u and v are connected in $G_{connected}$. If u does not delegate v, then u sends an check-interval message to v. Then either $u \in v.P^+ \cup v.S^+$ or v delegates u and u and v are weakly connected in $G_{connected}$.

If v is stored in an incoming message in $u.Ch$ then there can be the following cases:

- v is in a list in a list-update message. Then either v is stored in $u.P^+, u.S^+, u.S^-, u.P^-$ or v is delegated and u and v are weakly connected in $G_{connected}$ as shown in the proof of Lemma 4.3.11.
- v is a reference in an interval-update message, then either v is stored as a new reference for some data or if there is no corresponding data v is delegated to $w^-(v)$ (resp. $w^+(v)$) and u and v are weakly connected in $G_{connected}$ after $\mathcal{O}(1)$ rounds.
- v is a reference in an store-data message, then v is stored as a new reference in $u.DS$. And as already shown u and v are weakly connected after $\mathcal{O}(1)$ rounds.

□

Before we prove the Theorem 4.3.14 we introduce some additional definitions like in the analysis in the ATSS model.

Definition 4.3.9 *We define an* undirected path *p as a sequence of edges $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$, such that $\forall i \in \{1, \cdots, k\} : (v_i, v_{i-1}) \in E_{connected} \vee (v_{i-1}, v_i) \in E_{connected}$. We define $v_{min} = argmin\,\{h(v) : v \in p\}$ and $v_{max} = argmax\,\{h(v) : v \in p\}$. Then the* range of a path *$range(p)$ is given by $range(p) = v_{max} - v_{min}$.*

Lemma 4.3.13 *After $\mathcal{O}(1)$ rounds all nodes $u.S^*[i]$ and $u.S^*[i+1]$ (resp. $u.P^*[i]$ and $u.P^*[i+1]$) with $i < |u.S^*|$ will be connected and stay connected in every state after over nodes w' with $h(u.S^*[i]) < h(w') < h(u.S^*[i+1])$ (resp. $h(u.S^*[i]) > h(w') > h(u.S^*[i+1])$).*

Proof. In the periodic action u executes $Build - Triangle()$, in which u introduces every pair of nodes $u.S^*[i], u.S^*[i+1]$ to each other. The connecting path only changes if w.l.o.g. $u.S^*[i]$ delegates $u.S^*[i+1]$, but then $u.S^*[i]$ can delegate $u.S^*[i+1]$ only to a node w with $h(u.S^*[i]) < h(w) < h(u.S^*[i+1])$. By using this argument inductively $u.S^*[i]$ and $u.S^*[i+1]$ stay connected in every state after over nodes $w' : h(u.S^*[i]) < h(w') < h(u.S^*[i+1])$. □

We are now ready to prove Theorem 4.3.14.

Proof. We show that the stabilization time is in $\mathcal{O}(n)$. Let u_i and u_{i+1} be two arbitrary consecutive nodes in the sorted list. By Lemma 4.3.12 we know that in every graph after $\mathcal{O}(1)$ rounds there is a weakly connecting path for u_i and u_{i+1} in $G_{connected}$. By Lemma 4.3.13 we know that after $\mathcal{O}(1)$ further rounds all nodes $u.S^*[i]$ and $u.S^*[i+1]$ (resp. $u.P^*[i]$ and $u.P^*[i+1]$) with $i < |u.S^*|$ will be connected and stay connected in every state after over nodes $w' : h(u.S^*[i]) < h(w') < h(u.S^*[i+1])$. Then also Lemma 4.3.7 can be applied. We denote this state by s_{t_0}. We then show that the minimal range of all connecting paths is decreasing, such that the two nodes are directly connected after at most $\mathcal{O}(n)$ synchronous rounds.

Let p^t be a connecting path for two arbitrary consecutive nodes u_i and u_{i+1} at time t. We then show that we can find a connecting path at time t' $t' > t$ with a strictly smaller range. We firstly prove that if one of the border nodes v_{min} and v_{max} defining the range of p^t has two outgoing edges in $E^t_{connected} \cap p^t$, then we can construct a connecting path p^{t+1} with a smaller range. We will show this only for v_{min} as symmetric arguments can be applied for v_{max}.

If v_{min} has two outgoing edges $(v_{min}, u), (v_{min}, v) \in E^t_{connected} \cap p^t$ then in the next round according to Lemma 4.3.7 there is a directed path $p_1 = (v_{min} = w'_0, w'_1 = v_{min}.S^*[1], w'_2, \cdots w'_l = u)$ from v_{min} to u such that $h(v_{min}) \leq h(w'_i) \leq h(u) \forall w'_i$ and another path $p_1 = (v_{min} = w''_0, w''_1 = v_{min}.S^*[1], w''_2, \cdots w''_k = v)$ from v_{min} to v such that $h(v_{min}) \leq h(w''_i) \leq h(v) \forall w''_i$. Thus we can substitute the edges (v_{min}, u) and (v_{min}, v) by p_1 and p_2 leaving out v_{min} as p_1 and p_2 both include $v_{min}.S^*[1]$ and get a new weakly connecting path p_{t+1}.

It remains to show that it does not take to long before a border node has two outgoing edges. In case v_{min} does not have two outgoing edges we construct p^{t+1} out of p^t in the following way. If $(u, v) \in p^t$ with $h(u) < h(v)$ and v is delegated then according to Lemma 4.3.7 there is a directed path $p' = (u = w'_0, w'_1 = u.S^*[1], w'_2, \cdots w'_l = v)$ with $h(u) \leq h(w_i) \leq h(v)$ from u to v. Thus we substitute (u, v) by p' in p^{t+1}. If $(u, v) \in p^t$ with $h(u) < h(v)$ and v is not delegated then we simply keep $(u, v) \in p^{t+1}$. If $(u, v) \in p^t$ with $h(u) > h(v)$ and v is delegated then according to Lemma 4.3.7 there is a directed path $p' = (u = w'_0, w'_1 = u.P^*[1], w'_2, \cdots w'_l = v)$ with $h(u) \geq h(w'_i) \leq h(v)$ from u to v. Thus we substitute (u, v) by p' in p^{t+1}. If $(u, v) \in p^t$ with $h(u) > h(v)$ and v is not delegated, then u introduces itself to v and we substitute $(u, v) \in p^t$ by $(v, u) \in p^{t+1}$.

By using this construction scheme we show that it takes at most $\max\{0, n - (t - t_0)\}$ rounds before v^t_{min} has two outgoing edges for a path p^t. We prove this by induction on t. Let p^{t_0} be an arbitrary undirected path connecting two consecutive nodes u_i, u_{i+1} in the sorted list at time t_0. Then $v^{t_0}_{min}$ is the left border node of this path. If $v^{t_0}_{min}$ already has two outgoing edges on p^{t_0} then it takes obviously most $\max\{0, n\}$ rounds. Otherwise there is at least one incoming edge $(v, v^{t_0}_{min})$ on p^{t_0}. For each such incoming edge $(v, v^{t_0}_{min})$ the node v either introduces itself to $v^{t_0}_{min}$ or delegates $v^{t_0}_{min}$. In the first case $v^{t_0}_{min}$ then has an outgoing edge in p^1 according to our construction scheme. In the second case $(v, v^{t_0}_{min})$

is substituted by a path p' as described above. Then $v_{min}^{t_0}$ still has an incoming edge $(y', v_{min}^{t_0})$, but with $h(y') < h(v)$. Obviously $v_{min}^{t_0}$ can only be delegated n times before $(y', v_{min}^{t_0})$ is substituted by an outgoing edge.

Let p^t be an undirected path connecting u_i, u_{i+1} in the sorted list that was constructed out of p^{t_0} according to our construction scheme. If v_{min}^t already has two outgoing edges on p^t then it takes obviously most $\max\{0, n-(t-t_0)\}$ rounds. Otherwise there is at least one incoming edge (v, v_{min}^t) on p^t. Then $(v, v_{min}^t) \notin p^{t-1}$ but v_{min}^t was delegated to v by another node z with $h(z) > h(v)$, otherwise v would have introduced itself to v_{min}^t and v_{min}^t would have an outgoing edge instead. Thus $(z, v_{min}^t) \in p^{t-1}$. For this edge the same observation holds. As v_{min}^t has been delegated in every round before so $t - t_0$ times by nodes z with $h(z) > h(v)$ there are at most $n - (t - t_0)$ nodes w with $h(v_{min}^t) < h(w) < h(v)$ v_{min}^t can still be delegated to before the edge is substituted by an outgoing edge. If it takes at most $\max\{0, n-(t-t_0)\}$ rounds before v_{min}^t has two outgoing edges and we can construct a path with a smaller range then after $\mathcal{O}(n)$ rounds u_i, u_{i+1} are directly connected, i.e. $(u_i, u_{i+1}) \in E_e$ and $(u_{i+1}, u_i) \in E_e$. □

Theorem 4.3.15 *If G_{e-list}^t is a sorted list then after $\mathcal{O}(n)$ rounds the computation of P^{CONE} reaches a state $s_{t'}$ such that $G^{t'} \in CONE$.*

We will first prove that after $\mathcal{O}(n)$ rounds for every node u, $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$.

Lemma 4.3.14 *If G_{e-list}^t is a sorted list after $\mathcal{O}(n)$ rounds for every node u, $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$.*

Proof. According to Theorem 4.3.14 after $\mathcal{O}(n)$ rounds G_{e-list}^t is a sorted list. We then prove the lemma by induction on the capacities of the nodes. In fact we show the following hypothesis: If for all nodes v with $c(v) < c(u)$ $v.succ_1^+ = succ_1^+(v)$ and $v.pred_1^+ = pred_1^+(v)$ then in the next rounds $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$.

We start the induction on the node $u = v_{min}$ with the smallest capacity. Obviously for this node $succ(v_{min}) = succ_1^+(v_{min})$ and $pred(v_{min}) = pred_1^+(v_{min})$. As G_{e-list}^t is a sorted list also $v_{min}.S^*[1] = succ(v_{min})$ and $v_{min}.P^*[1] = pred(v_{min})$. Then v_{min} is already connected to $succ_1^+(v_{min})$ and $pred_1^+(v_{min})$ and $v_{min}.succ_1^+ = succ_1^+(v_{min})$ and $v_{min}.pred_1^+ = pred_1^+(v_{min})$.

In the inductive step let u be a node for which for all nodes v with $c(v) < c(u)$ $v.succ_1^+ = succ_1^+(v)$ and $v.pred_1^+ = pred_1^+(v)$. Let $z = succ_1^+(u)$. Then let $v_1, v_2 \cdots v_l$ be the nodes in the sorted list between u and z. We know that for each v_i $c(v_i) < c(u)$ and thus $v_i.succ_1^+ = succ_1^+(v_i)$ and $v_i.pred_1^+ = pred_1^+(v_i)$. Let $v_j = \text{argmax}\{c(v_i) : i \leq l\}$. Then for v_j $succ_1^+(v_j) = z$ and $pred_1^+(v_i) = u$. Thus in the next round v_j will introduce z to u and vice versa in the Build-Triangle() action and $u.succ_1^+ = succ_1^+(u) = z$. By symmetric arguments we can show that also $u.pred_1^+ = pred_1^+(u)$. □

Lemma 4.3.15 *If G_{e-list}^t is a sorted list and for every node u $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$ then after $\mathcal{O}(n)$ rounds $u.S^+ = S^+(u)$ and $u.P^+ = P^+(u)$ and $u.S^- = S^-(u)$ and $u.P^- = P^-(u)$.*

Proof. We begin by showing that $u.S^- = S^-(u)$ and $u.P^- = P^-(u)$. If G_{e-list}^t is a sorted list and for every node u $u.succ_1^+ = succ_1^+(u)$ and $u.pred_1^+ = pred_1^+(u)$ then in the next round each node u will receive a message $m = (build - triangle, v)$ from each node v with $v.succ_1^+ = succ_1^+(v) = u$ or

$v.pred_1^+ = pred_1^+(v) = u$. Then $u.S^- = S^-(u)$ and $u.P^- = P^-(u)$. We show that after $\mathcal{O}(n)$ further rounds $u.S^+ = S^+(u)$ and $u.P^+ = P^+(u)$ by induction on the capacity of the nodes. We prove the following hypothesis: If for all nodes v with $c(v) > c(u)$ $v.S^+ = S^+(v)$ and $v.P^+ = P^+(v)$ then in the next round $u.S^+ = S^+(u)$ and $u.P^+ = P^+(u)$. We start the induction on the node u_{max} with the greatest capacity. Obviously for u_{max} $u_{max}.S^+ = S^+(u_{max})$ and $u_{max}.P^+ = P^+(u_{max})$. In the inductive step let u be a node for which for all nodes v with $c(v) > c(u)$ $v.S^+ = S^+(v)$ and $v.P^+ = P^+(v)$. Then we know by the first part of the proof that $u \in u.succ_1^+.P^-$ and thus in the next round $u.succ_1^+$ sends a message $m = (list - update, u.succ_1^+.S^+ \cup \{u.succ_1^+\})$. As $u.succ_1^+.S^+ = S^+(succ_1^+(u))$ also $u.S^+ = S^+(u)$. By symmetric arguments we can show that $u.P^+ = P^+(u)$. □

Combing Lemma 4.3.14 and Lemma 4.3.15 Theorem 4.3.15 follows immediately.

Combining Lemma 4.3.11, Theorem 4.3.14 and Theorem 4.3.15 we can show that Theorem 4.3.13 holds, and by our protocol each weakly connected network converges to a CONE network in $\mathcal{O}(n)$ rounds.

Like in the analysis in the ATSS model it remains to show that not only the correct topology is formed but that also each node stores the correct data. We show that after $\mathcal{O}(\log n)$ further rounds w.h.p. each node stores the correct data.

Theorem 4.3.16 *If in an initial graph $G \in CONE$ every nodes executes the protocol $P^{CONEsync}$ then each possible computation leads to a graph $G' \in CONE$.*

Proof. The proof is the same as in the asynchronous setting for P^{CONE}. □

Theorem 4.3.17 *If $G \in CONE$ after $\mathcal{O}(\log n)$ rounds w.h.p. each node stores exactly the data it is responsible for.*

Proof. According to Theorem 4.3.2 each node knows which node is responsible for parts of the interval it supervises. In our described algorithm each node u checks whether it is responsible for the data it currently stores by sending a message to the node v that u assumes to be supervising the corresponding interval. If v is supervising the interval and u is responsible for the data, then u simply keeps the data. If v is not supervising the data or u is not responsible for the data then v sends a reference to u with the id of a node that v assumes to be supervising the interval. Then u forwards the data to the new reference and does not store the data. By forwarding the data by Greedy Routing it reaches a node supervising the corresponding interval in $\mathcal{O}(\log n)$ rounds according to Lemma 4.3.3, this node then tells the responsible node to store the data. Thus after $\mathcal{O}(\log n)$ rounds all data is stored by nodes that are responsible for the data. □

Stabilization Work

Theorem 4.3.18 *If in an initial graph $G \in IT$ every nodes executes the protocol $P^{CONEsync}$ then each possible computation leads to a graph $G' \in CONE$ with a stabilization work of $\mathcal{O}(n^2 + Dn \log n)$ w.h.p. if there are D data items stored in the network.*

Before we prove this theorem, we state some generalized versions of Lemma 4.3.1 and Lemma 4.3.2.

Lemma 4.3.16 *At any time of the computation for each $u \in V$ in a valid state, $|u.S^+|$ and $|u.P^+| \in \mathcal{O}(\log n)$ w.h.p..*

Lemma 4.3.17 *At any time of the computation for each $u \in V$ in a valid state, $|u.S^-|$ and $|u.P^-| \in \mathcal{O}(\log n)$ w.h.p..*

These lemmas follow directly from the proofs of Lemma 4.3.1 and Lemma 4.3.2, as the arguments hold for arbitrary sets of nodes left or right from a node u. We are now ready to prove the theorem.

Proof. Each node u receives at most $\mathcal{O}(\log n)$ introduction messages w.h.p. by nodes v currently in $u.P^*$ and $u.S^*$ according to Lemma 4.3.17. Furthermore each node only receives at most $\mathcal{O}(1)$ messages from nodes w' that do not belong to (the updated) $u.P^*$ and $u.S^*$, as u responds to such nodes by sending a forward message containing a node w'' between u and w', such that w' will possibly introduce itself to w'' but not u. A node only sends introduction messages to nodes in $u.P^*$ and $u.S^*$. Thus in total each node sends and receives $\mathcal{O}(n \log n)$ introduction messages w.h.p..

Each node receives at most $\mathcal{O}(1)$ list-update messages by $u.p_1^+$ and $u.s_1^+$. Furthermore u only receives at most $\mathcal{O}(1)$ list-update messages from nodes w' that are not (the updated) $u.p_1^+$ and $u.s_1^+$, as u responds to such nodes by sending a forward message containing a node w'' between u and w', such that w' will send its list-update message possibly to w'' but not u. A node only sends list-update messages to nodes in $u.P^-$ and $u.S^-$. As each list-update message contains up to $\mathcal{O}(\log n)$ identifiers w.h.p., we count them as $\mathcal{O}(\log n)$ messages. Then each node sends and receives w.h.p. $\mathcal{O}(n \log^2 n)$ list update messages.

Each node u sends a check-interval message for each interval it seems to be responsible for to the corresponding reference node, that should supervise the interval. There can be at most n reference nodes, thus a node sends at most $\mathcal{O}(n^2)$ check-interval messages.

A node receives a check-interval message for an interval it does not supervise at most once, i.e. if the sending node is not in $u.S^+$ or $u.P^+$. Furthermore a node receives at most one check-interval message from another node on one round. Thus a node u receives at most $\mathcal{O}(n \log n)$ check-interval messages w.h.p. according to Lemma 4.3.16 in total, as each sending node has to be currently in $u.S^+$ or $u.P^+$.

For each received check-interval message a node sends at most one update-interval message. Thus a node sends at most $\mathcal{O}(n \log n)$ update-interval messages in total w.h.p.. A node receives at most one update-interval message by another node. An update-interval message is only received in response to a check-interval message. As u sends at most $\mathcal{O}(n^2)$ check-interval messages, it can also only receive at most $\mathcal{O}(n^2)$ update-interval messages in total. Thus a node u receives and sends at most $\mathcal{O}(n^2)$ update-interval messages w.h.p. in total.

We can also show that each node u sends and receives at most $\mathcal{O}(n^2)$ forward messages w.h.p.. W.l.o.g. we show that a node u receives at most $\mathcal{O}(n^2)$ forward messages containing nodes w', such that $h(u) < h(w')$. There are three cases in which u receives such a message. Either u sent an introduction message to a node z with $h(w') < h(z)$ and received w' in response or w' is sent to u by a node y that simply forwards w' to a closer node according to its local sorting or u received w' in a list-update message. Obviously the first case and the last case happens at most $\mathcal{O}(n)$ times and each list-update message contains up to $\mathcal{O}(\log n)$ ids. Thus u receives at most $O(n \log n)$ ids by list-update messages or responses to introduction messages. A node receives at most $\mathcal{O}(1)$ forward messages from other nodes forwarding some ids. Thus by the second case a node u receives at most $\mathcal{O}(n^2)$ forward messages. For each received forward message a node sends at most $\mathcal{O}(1)$ forward messages. Thus in total each node sends and receives at most $\mathcal{O}(n^2)$ forward messages.

The number of forward-data and store-data messages are depending on the number of stored data items in the network. Let D be the number of data items, then we can show that each node sends and receives at most $\mathcal{O}(Dn \log n)$ forward-data and store-data messages w.h.p.. A forward data-message is sent if a node receives a forward-data message for a data item that it is not supervising or if the node no longer

stores the data item, i.e. a update-interval message is received. Due to the greedy routing approach a node receives a forward-data message at most once for each data item, that it is not supervising. If the node is supervising the data item it receives at most $\mathcal{O}(n)$ forward-data messages in case that the responsible node changes $\mathcal{O}(n)$ times. As we have seen a node receives at most $\mathcal{O}(n \log n)$ update-interval messages w.h.p. thus a node sends at most $\mathcal{O}(n \log n)$ forward-data messages w.h.p. for a data item. A store-data message is received at most $\mathcal{O}(n)$ times for a data item, in case that the supervising node for this data item changes $\mathcal{O}(n)$ times. A node also sends a store-data message at most $\mathcal{O}(n)$ times for one data item, in case that the responsible node changes up to $\mathcal{O}(n)$ times. In total each node sends and receives at most $\mathcal{O}(Dn \log n)$ forward-data and store-data messages w.h.p.. □

Maintenance Work

Theorem 4.3.19 *If in an initial graph $G \in CONE^*$ every node executes the protocol $P^{CONEsync}$ then each possible computation leads to a graph $G' \in CONE$ with a maintenance work of $\mathcal{O}(r + \log^2 n)$ w.h.p. if a node stores data items supervised by r reference nodes.*

Proof. Each node u sends $\mathcal{O}(1)$ forward/introduction messages to each node in $u.S^-$ and $u.P^-$. Additionally u sends $\mathcal{O}(1)$ forward/introduction messages to $u.pred_1^+$ and $u.succ_1^+$. Thus according to Lemma 4.3.2 a node u sends $\mathcal{O}(\log n)$ forward/introduction messages w.h.p.. A node u receives $\mathcal{O}(1)$ forward/introduction messages from each node in $u.S^-$ and $u.P-$ and from $u.pred_1^+$ and $u.succ_1^+$. Thus again according to Lemma 4.3.2 a node u receives $\mathcal{O}(\log n)$ forward/introduction messages w.h.p..

Each node u sends its lists $u.S^+$ (resp. $u.P^+$) to all nodes in $u.P^-$ (resp. $u.S^-$). As each list $u.S^+$ (resp. $u.P^+$) is of size $\mathcal{O}(\log n)$ w.h.p. according to Lemma 4.3.1 we count sending one list as sending $\mathcal{O}(\log n)$ messages. Then according to Lemma 4.3.2 a node sends $\mathcal{O}(\log^2 n)$ list-update messages. Each node u receives a list-update message from $u.s_1^+$ and $u.p_1^+$ thus according to Lemma 4.3.1 each node receives $\mathcal{O}(\log n)$ list-update messages.

Each node u sends a check-interval message for each interval it seems to be responsible for to the corresponding reference node, that should supervise the interval. Thus each node sends at most r check-interval messages if it stores r reference nodes. Each node receives at most $\mathcal{O}(\log n)$ check-interval messages w.h.p., as each sending node has to be in $u.P^+$ or $u.S^+$. According to Theorem 4.3.17 each node already stores the correct data, thus no update-interval messages, no forward-data and no store-data messages are sent. Thus in total the maintenance work of each node is $\mathcal{O}(r + \log^2 n)$ w.h.p.. □

Single Join and Leave Event and Capacity Change

Concerning the network operations in the network, i.e. the joining of a new node, the leaving of a node and the capacity change of a node, we show the following:

Theorem 4.3.20 *In case a node u joins a stable CONE network, or a node u leaves a stable CONE network or a node u in a stable CONE network changes its capacity, we show that in any of these three cases it takes $\mathcal{O}(\log n)$ rounds and $\mathcal{O}(\log^3 n)$ messages to recover to the new stable state.*

We show the statement by considering the 3 cases separately. We start with the case of a joining node.

Theorem 4.3.21 *If a node u joins a stable CONE network it takes $\mathcal{O}(\log n)$ rounds and $\mathcal{O}(\log^3 n)$ messages to recover to the new stable state.*

Proof. If u joins the network u is connected to a node v already in the network by an edge (u, v), i.e. v is either stored in $u.S^*[1]$ or $u.P^*[1]$. In the Build-Triangle() action u then introduces itself to v after one round. Then v either stores u in its internal variable or delegates u in the Build-Triangle(u) action. Even if v stores u u is also introduced to nodes that should also know u in the Build-Triangle() action. By these delegations and introduction u is forwarded according to the CONE Greedy routing we introduced in Definition 4.3.2. Then according to Lemma 4.3.3 after $\mathcal{O}(\log n)$ rounds w.h.p. u is connected to a node w such that $w = succ(u)$ or $w = pred(u)$, i.e. $u.S^*[1] = w$ and $w.P^*[1] = u$ or $u.P^*[1] = w$ and $w.S^*[1] = u$. W.l.o.g. we assume $w.S^*[1] = u$. Let $x = w.P^*[1]$ before the integration of u, then w introduces x to u, as soon as w is connected to u. Thus u is connected to $succ(u)$ and $pred(u)$ after $\mathcal{O}(\log n)$ rounds w.h.p..

Now there can be two cases either $w = pred_1^+(u)$ or $v \in u.P^-$. In the first case u learns $u.P^+$ by receiving a list-update from w. In the second case $pred_1^+(u) \in w.P^+$ according to Theorem 4.3.6. Then $u = succ_1^+(w) = w.succ_1^+$. Then according to the protocol $P^{CONEsync}$ w introduces u to $w_1 = w.pred_1^+$. Then again there can be two cases either $w_1' = pred_1^+(u)$ or $w_1 \in S^-(u)$. Thus we can apply the same arguments inductively until u is connected to a node w_i with $w_i = pred_1^+(u)$, or $P^+(u)$ is empty. According to Lemma 4.3.1 it takes at most $\mathcal{O}(\log n)$ rounds w.h.p. until w_i is connected to u. Then in the next round also $u.P^+ = P^+(u)$ by receiving a list-update from w_i. Then all w_j with $j < i$ are in $u.P^-$ as $P^-(u)$ by definition is a subset of $P^+(w)$ before the integration of u. Thus also $u.P^- = P^-(u)$. As u is also connected to $x = succ(u)$ we can show by symmetric arguments that after $\mathcal{O}(\log n)$ rounds w.h.p. also $u.S^+ = S^+(u)$ and $u.S^- = S^-(u)$.

As soon as u is connected to $pred_1^+(u)$ and $succ_1^+(u)$ also $y.P^- = P^-(y)$ and $y.S^- = S^-(y)$ for all nodes $y \in V$. It remains to show that also $y.P^+ = P^+(y)$ and $y.S^+ = S^+(y)$ for all nodes $y \in V$. If for a node y $P^+(y)$ (resp. $S^+(y)$) changes after u joins the network, then $u \in y.P^+(u)$ (resp. $u \in S^+(y)$) after the join of u. W.l.o.g we assume that $P^+(y)$ changes.

Then according to our analysis above there is a node z in $y.P^+$ with $z.pre_1^+ = u$ and $z.P^+ = P^+(z)$. Then z sends a list-update message to all nodes in $z.S^-$. As $z \in y.P^+$ $z.S^- \cap (y.P^+ \cup y) \neq \emptyset$ and for at least two nodes in $y.P^+$ the P^+ lists are correct. By applying this argument inductively after $\mathcal{O}(\log n)$ rounds w.h.p. according to Lemma 4.3.1 all nodes in $P^+(y)$ hold the correct P^+ lists and then also y receives a list-update from $pre_1^+(y)$ and $y.P^= P^+(y)$. By symmetric arguments we can also show that after $\mathcal{O}(\log n)$ rounds w.h.p. $y.S+ = S^+(y)$ for all nodes $y \in V$. Thus after $\mathcal{O}(\log n)$ rounds a stable state is reached and $G' \in CONE$.

As long as u is not connected to w and x, u is only connected to v thus u sends and receives at most $\mathcal{O}(1)$ messages w.h.p. in each round during this time. Afterward u is only connected to nodes it will also store in the stable state. Thus according to Theorem 4.3.19 u will only send and receive $\mathcal{O}(\log^2 n)$ messages in each round w.h.p.. Then in total u sends and receives at most $\mathcal{O}(\log^3 n)$ messages w.h.p.. Each node $y \in V$ for which the sets of stored nodes do not change also sends and receives at most $\mathcal{O}(\log^3 n)$ messages w.h.p. according to Theorem 4.3.19 as they only receive at most one forward message containing u and send one message delegating u in each round additionally. Nodes for which the sets of stored nodes change by the integration of u receive the correct lists and delegate all nodes not longer stored once thus each node sends and receives $\mathcal{O}(\log^2 n)$ messages in total additionally. Summing up each node sends and receives at most $\mathcal{O}(\log^3 n)$ messages until a node u is integrated and the computation reaches a stable state. □

We proceed with the case of a leaving node.

Theorem 4.3.22 *If a node u leaves a stable CONE network it takes $\mathcal{O}(\log n)$ rounds and $\mathcal{O}(\log^3 n)$ messages to recover to the new stable state.*

Proof. The proof is similar to the proof in the case of a joining node. We assume that u is not the node with the greatest capacity, otherwise the network gets disconnected when u leaves.We further assume that u has a left and right neighbor, i.e. u is not the node with the maximal or minimal position, otherwise the network remains stable even if u leaves. In the following let $P^+(x)$ denote the set before the leaving of u and $P'^+(x)$ the set after the leaving of u.

If u leaves the network, then for all nodes $v \in S^-(u)$ $pred_1^+(v)$ changes and for all nodes $w \in P^-(u)$ $succ_1^+(w)$ changes. Then $pred_1'^+(v) \in P^*(u)$ and $succ_1'^+(w) \in S^*(u)$. Thus for nodes x in $P^*(u)$ and $S^*(u)$ $S^-(x)$ resp. $x.P^-(x)$ can change. Furthermore like in the case of a joining node for nodes y with $u \in P^+(y)$ or $u \in S^+(y)$ the sets $P^+(y)$ resp. $S^+(y)$ can change.

As u is not the node with the maximal capacity at least one of the sets $u.P^+$ and $u.S^+$ is not empty. W.l.o.g. let $u.P^+$ not be empty, then each node v in $u.S^-$ is connected to $u.pred_1^+$. We now show that for each $v \in u.S^-$ after $\mathcal{O}(\log n)$ round w.h.p. $v.P^+ = P'^+(v)$. Let $l = |u.P^*|$ then if all nodes v are connected to $w = pred_1^+(u) = u.P^*[l]$ either $v \in S'^-(w)$ and w stores v or w delegates v to $u.P^*[l-1]$. Thus by applying the same argument inductively until after $\mathcal{O}(\log n)$ rounds w.h.p. for each $v \in u.S^-$ $v.pred_1^+ = pred_1'^+(v) = u.P^*[i]$ for some $i \leq l$. Then in the next round $v.P^+ = P'^+(v)$. We now show that for each $v \in u.S^-$ after $\mathcal{O}(\log n)$ round w.h.p. also $v.P^- = P'^-(v)$. We already showed that for each $v \in u.S^-$ $v.pred_1^+ = pred_1'^+(v) = u.P^*[i]$. Let $w = v.pred_1^+ = u.P^*[i]$ Then w integrates all v with $w = v.pred_1^+$ in $w.S^-$. If there are nodes $v' \in u.P^-$ with $succ_1'^+(v') = v$, i.e. $v' \in P'^-(v)$, then also $u.P^*[i-1] \in P'^-(v)$ and $w = u.P^*[i]$ forwards v to $u.P^*[i-1]$. By applying this argument iteratively on the $u.P^*[j]$s according to Lemma 4.3.2 after $\mathcal{O}(\log n)$ rounds w.h.p. $v.P^- = P^-(v)$ for all $v \in u.S^-$. As for all nodes $v \in u.S^*$ $v.P^- = P^-(v)$ and $v.P^+ = P^+(v)$, also $w.P^- = P^-(w)$ and $w.P^+ = P^+(w)$ for all nodes $w \in u.P^*$. If for all nodes $v \in u.S^*$ $v.P^- = P^-(v)$, then for all $w \in u.P^*$ $w.succ_1'^+ = succ_1'^+(w)$, as their successor has to be a node in $u.S^*$. If for all nodes $v \in u.S^*$ $v.P^+ = P^+(v)$, then for all $w \in u.P^*$ $w.S'^- = S'^-(w)$, as only nodes in $u.S^*$ might be added. Then for all nodes $y \in V$ after at most $\mathcal{O}(\log n)$ rounds $y.S'^- = S'^-(y)$ and $y.P'^- = P'^-(y)$. Like in the case of a joining node it remains to show that $y.S'^+ = S'^+(y)$ and $y.P'^+ = P'^+(y)$ for all nodes $y \in V$. If for a node y $P^+(y)$ (resp. $S^+(y)$) changes after u leaves the network, then $u \in P^+(y)$ (resp. $u \in S^+(y)$) after the join of u. W.l.o.g we assume that $P^+(y)$ changes. Then according to our analysis above there is a node z in $y.P^+$ with $z.pred_1^+ = u$ and $z.P'^+ = P'^+(z)$. Then z sends a list-update message to all nodes in $z.S^-$. As $z \in y.P^+$ $z.S^- \cap (y.P^+ \cup \{y\}) \neq \emptyset$ and for at least two nodes in $y.P^+$ the P^+ lists are correct. By applying this argument inductively after $\mathcal{O}(\log n)$ rounds w.h.p. according to Lemma 4.3.1 all nodes in $P^+(y)$ hold the correct P^+ lists and then also y receives a list-update from $pred_1^+(y)$ and $y.P'^= P'^+(y)$. By symmetric arguments we can also show that after $\mathcal{O}(\log n)$ rounds w.h.p. $y.S'^+ = S'^+(y)$ for all nodes $y \in V$. Thus after $\mathcal{O}(\log n)$ rounds w.h.p. a stable state is reached and the resulting graph $G' \in CONE$. Each node $y \in V$ for which the sets of stored nodes do not change also sends and receives at most $\mathcal{O}(\log^3 n)$ messages w.h.p. according to Theorem 4.3.19. Nodes for which the sets of stored nodes change by the leaving of u receive the correct lists and delegate all nodes they don't store at most once thus each node sends and receives $\mathcal{O}(\log^2 n)$ messages w.h.p. in total additionally. Summing up each node sends and receives at most $\mathcal{O}(\log^3 n)$ messages w.h.p. before the computation reaches a stable state. □

If the capacity of a single node u in a stable CONE network changes we can apply the same arguments as for the cases of a joining or leaving node. By its local knowledge u can correct its lists $u.S^+$, $u.S^-$,

$u.P^+$. and $u.P^-$ in one single round. For all other nodes $y \in V - \{u\}$ we can use the arguments of the case of a joining node if $c(u)$ increases or the case of a leaving node if $c(u)$ decreases. Thus the following theorem follows.

Theorem 4.3.23 *If a node u in a stable CONE network changes its capacity it takes $\mathcal{O}(\log n)$ rounds and $\mathcal{O}(\log^3 n)$ messages w.h.p. to recover to the new stable state.*

4.4 Conclusion & Outlook

This chapter presented the application of topological self-stabilization to two distributed hash tables. We introduced Re-Chord, a Chord like distributed hash table that can be formed by a self-stabilizing protocol $P^{RE-CHORD}$ with a stabilization time of $\mathcal{O}(n \log n)$, a stabilization work of $\mathcal{O}(n \log^3 n)$, and a maintenance work per round of $\mathcal{O}(\log^2 n)$ for each node. It seems like that by a modified analysis and maybe some modifications of the protocol a sublinear stabilization time can be shown like in the analysis in [29]. This results from the observation that if a node u^i forwards a node v to a virtual sibling u^j the distance to v is halved. Thus a polylogarithmic stabilization time seems likely. Then also the stabilization work could be improved.

This chapter also presented a heterogeneous distributed hash table for which we considered heterogeneity in terms of different capacities of the nodes. In the presented overlay network CONE-DHT each node has a degree of $\mathcal{O}(\log n)$ w.h.p.. The protocol P^{CONE} for this overlay network is correct in the ATSS and STSS model. In the STSS model we showed a stabilization time of $\mathcal{O}(n)$, a stabilization work of $\mathcal{O}(n^2 + Dn \log n)$ w.h.p. if there are D data items stored in the network, and a maintenance work of $\mathcal{O}(r + \log^2 n)$ w.h.p. if a node stores data items supervised by r reference nodes. This is the first attempt to present a self-stabilizing method for a heterogeneous overlay network and it works efficiently regarding the information stored in the hosts. Furthermore our solution provides a low degree, fair load balancing and polylogarithmic update cost in case of joining or leaving nodes. Our solution is independent of the capacities and their distribution, thus there are no restrictions on the minimal or maximal capacity of a node. The properties of the overlay network even hold for arbitrary height functions as long as they are monotonic.

Also for the CONE-DHT the optimality of the stabilization time and work remains an open question. Also for P^{CONE} it might be possible to exploit the logarithmic degree of the nodes and logarithmic length of a routing path in the analysis to show that the stabilization time is polylogarithmic. A natural extension of this approach is to consider heterogeneity in more than one aspect, e.g. nodes with heterogeneous capacities and bandwidth. Then it could be necessary to build a 2-dimensional overlay network that supports heterogeneity. Even heterogeneity in further aspects like stability or trustfulness of the nodes could be considered to build overlay networks for specific applications.

CHAPTER 5

Prefix and Range Queries on Distributed Hash Tables

5.1 Introduction

So far the distributed hash tables we presented only supported exact queries, i.e. on a look-up request the system answers with the requested data item, if it is currently stored in the system or gives a negative output if the data item is not found. In section 5.2 we present a solution that can be built on top of such distributed hash tables and enables them to also support prefix queries. If an item with the searched key is not found the system responds with an item that has a similar key - in this case the longest matching prefix - like the key of the searched item. We therefore introduce a data structure called hashed Patricia trie, that is able to support prefix queries on top of distributed hash tables. This data structure supports the following operations: *PrefixSearch(x)* and *Insert(x)* can be executed in $\mathcal{O}(\log(|x|))$ hash table accesses while *Delete(x)* only needs a constant number of hash table accesses.

We will then extend this solution to also support range queries on the keys by the hashed predecessor Patricia trie in section 5.3. We get a hashed predecessor Patricia trie out of our hashed Patricia trie by adding pointers, pointing to the minimal and maximal key in a sub trie. For this data structure we show that all operations *PredecessorSearch(x)*, *Insert(x)* and *Delete(x)* can be executed in $\mathcal{O}(\log \log U)$ hash table accesses when U is the size of the universe of the keys.

5.1.1 Related Work

The design of efficient search structures is a classical problem in computer science. The longest prefix problem we consider in the first section can be defined as follows. From a set S of keys, represented as binary strings, find a key $y \in S$ that has the longest common prefix with the search key x. The predecessor problem then is a search problem that is closely related to the longest prefix problem. In a predecessor problem we are given a key and the problem is to find $y = \max\{z \in S \mid z \leq x\}$. In our context with strings, a natural interpretation of $z \leq x$ is that z is lexicographically smaller than x (i.e., the first letter different in x and z is lexicographically smaller in z than in x). In this case, either the closest predecessor or the closest successor of x has the longest prefix match with x.

When using standard balanced search trees to solve the longest prefix problem like in [5] or the related problem of predecessor search on $n = |S|$ keys, the worst case execution time is in $\mathcal{O}(\log n)$ due to traversing the edges of the tree. Patrascu et al. [63] gave tight bounds of $\Theta(\log l)$ for the runtime of predecessor search on a data structure in a RAM model with words of length $l = \Theta(\log n)$. The upper bound results from using the van Emde Boas data structure presented in [76]. The van Emde Boas tree

comes with a specific layout limiting its use to a small range of values to be space efficient. When hashing its entries as we do it with our extended Patricia trie, then the van Emde Boas tree can be used to run *PrefixSearch(x)*, *Insert(x)* and *Delete(x)* in time $\Theta(\log |x|)$ in the worst case (w.h.p., where the probability is due to the hashing). However achieving a constant runtime for *Delete(x)* and a constant number of data structure updates for each operation would require major modifications in the van Emde Boas tree which is already a non-trivial, recursively defined data structure. We believe that our approach is much more intuitive and simple to understand.

A *trie* is a search tree in which every edge represents a letter and every node represents a string consisting of the letters of the unique path from the root to that node. Thus the depth of a trie for a set of strings S is equal to the maximum length of a string in S and therefore can be much larger than the depth of a balanced search tree. In the presented solution we will use a Patricia trie, which is a compressed trie, i.e. edges do not represent letters but strings and a path in a trie can be compressed to an edge in the Patricia trie. A formal definition will be given later.

Trie hashing has been used in [44] in an approach close to the hardware level called HEXA to reduce the memory needed to store a node of a trie and its information by avoiding string pointers to the next node. This approach is validated for IP look-up in routers and string matching. Our approach follows the idea of HEXA in some points, as we only store labels of the edges and no string pointers. But since we are concentrating on a higher level by using our data structure in a distributed setting like a DHT, we focus on the number of DHT accesses instead of minimizing the computations and memory usage, although our approach is asymptotical optimal in the memory usage.

Trie hashing has also been used, for example, in file systems [48, 49], IP look-up [77] and distributed hash tables [66] in order to provide efficient longest prefix searching. While the update cost of these data structures is expensive in the worst case (i.e., linear in the size of the key), Waldvogel et al. [77] as well as Ramabhadran et al. [66] use binary search on the prefix length to obtain a runtime of $\mathcal{O}(\log |x|)$ resp. limit the number of messages to $\mathcal{O}(\log |x|)$ for *PrefixSearch(x)*.

The approach of Ramabhadran et al. called Prefix Hash Tree [66] is probably most comparable to ours. Both are based on Trie hashing and both are designed for DHT-based peer-to-peer systems. The main advantages of our hashed Patricia trie compared to the Prefix Hash Tree are the minimized update costs. Using the Prefix Hash Tree, insertions and deletions can lead to worst case costs of $\mathcal{O}(|\text{max. key length}|)$. This is due to the buckets that contains the keys. If the number of keys stored in a bucket rises beyond its maximum size the bucket is split, and it might be split up to $\mathcal{O}(|\text{max. key length}|)$ times (when all keys are distributed to the same child). The same problem occurs in case of the deletion of a key. Ramabhadran et al. present a workaround by allowing the maximum size of the buckets to be exceeded. A Patricia trie is a compressed trie and thus contains less nodes in average, so less hash table entries are needed in our solution. In a hashed Patricia trie keys of arbitrary lengths can be stored instead of a fixed length in the Prefix Hash Tree. However, notice that the Prefix Hash Tree also supports range queries, which are not considered in this paper.

A better solution for prefix search in distributed systems is due to Awerbuch and Scheideler [3]. They embed a skip graph [2] based on the entries in S into a distributed system using consistent hashing and store the addresses of the nodes for every link of the skip graph so that each link traversal in the skip graph just means a single message hop in the distributed system. While this allows to execute *PrefixSearch(x)*, *Insert(x)* and *Delete(x)* in $O(\log n)$ communication rounds, the update cost when nodes join and leave the system can be much larger than in a DHT. With the hashed Patricia trie we minimize the update costs and furthermore present a data structure applicable on any kind of DHT that provides write and read commands.

In the second part of this chapter we will consider the predecessor problem to be able to support range queries for the key set stored in a distributed hash table. In [6] and [1] an overview of the different approaches for solving the predecessor problem is given. Using exponential search trees one can achieve the following runtime only depending on the number of key currently in the data structure while only using linear space: $\mathcal{O}(\min\left\{\log\log n + \log n/\log\log U, \log\log n * \frac{\log\log U}{\log\log\log U}, \right\})$. Thus if $n = \mathcal{O}(\log U)$ this solution outperforms our solution, but if $n = \mathcal{O}(U)$ our solution is better and in fact optimal.

We will reuse some techniques used for the x- and y-fast tries [80] introduced by Willard in our solution and combine them with the space efficient Patricia trie. Y-fast tries are also space efficient but have the drawback that they consist of a combination of different data structures, the keys are stored in several balanced search trees and for every search tree a representative is held in a x-fast trie. We therefore think that our solution using only a single Patricia trie is much more intuitive and simple to understand. Furthermore for y-fast tries only an amortized update cost of $\mathcal{O}(\log\log U)$ can be shown, whereas for the hashed Predecessor Patricia trie a worst case update cost of $\mathcal{O}(\log\log U)$ can be shown.

5.2 Hashed Patricia Trie: Efficient Longest Prefix Queries on Distributed Hash Tables

5.2.1 Introduction

In this section we consider the problem of longest prefix queries on distributed hash tables. In the following we formally define the problem and present a data structure that efficiently supports longest prefix queries with the help of any common DHT, e.g. Chord [75] or Pastry [70].

Definition 5.2.1 (Longest Prefix Problem) *Given a set of binary strings S and a binary string x, find $y \in S$ such that y has the longest common prefix with x of all strings in S. The prefix of a string is a sub string beginning with the first bit. If there are several keys with a longest prefix, any one of them is returned as y.*

In order to provide a data structure for the problem of longest prefix queries we need the following operations:

Insert(x): this adds the key x to the set S. If x already exists in S, it will not be inserted a second time.

Delete(x): this removes the key from the set S, i.e., $S := S - \{x\}$.

PrefixSearch(x): this returns a key $y \in S$ that has a longest common prefix with x.

We will present a data structure called *hashed Patricia trie* for these operations, which is based on an extension of the Patricia trie [56] over the data set embedded into a hash table. With this data structure, *PrefixSearch(x)* and *Insert(x)* can be executed in $\mathcal{O}(\log(|x|))$ hash table accesses while *Delete(x)* only needs a constant number of hash table accesses. Moreover, the number of changes in the data structure for each *Insert(x)* and *Delete(x)* operation is a constant. That is, the costs of the presented operations are independent of the size of the data structure as they only depend on the length of the input $|x|$. Our bounds on the hash table accesses imply that only few messages have to be sent when using the well-known distributed hash table (DHT) approach in order to realize our data structure. Most importantly just $\mathcal{O}(1)$ updates in the DHT are needed per operation, so it is easy to keep the DHT synchronized. We further point out that our solution is asymptotically optimal in the use of memory space, requiring only

Θ(sum of all key lengths) of memory. This implies that the data structure can easily be implemented on top of a DHT without producing an overhead in terms of memory usage.

5.2.2 Our contributions

We present a data structure that supports longest prefix queries on distributed hash tables and give the costs of the operations *PrefixSearch(x)*, *Insert(x)*, and *Delete(x)*, given in theorems 5.2.3- 5.2.5. These results are based on work published in [40]:

Sebastian Kniesburges, Christian Scheideler, Hashed Patricia Trie: Efficient Longest Prefix Matching in Peer-to-Peer Systems, In the 5th Workshop on Algorithms and Computation 2011 (WALCOM 2011), New Delhi, India, February, 2011.

5.2.3 Hashed Patricia Trie

In this section we describe our hashed Patricia trie. We first review the structure of an ordinary Patricia trie, then describe its modifications, and finally show how to combine it with a hash table.

Patricia Trie

The concept of a Patricia trie was first proposed by Morrison in [56]. Suppose for the moment that all keys in the given key set K have the same length (i.e., consist of the same number of letters). In this case, the Patricia trie of these keys is a compressed trie in which every edge now represents a string (instead of just a single letter), every inner node (except for the root) has two children, and every node represents a string resulting from the concatenation of the strings along the unique path from the root to that node.

See Figure 5.1 for an example.

In the following, we will restrict ourselves to keys representing binary strings. For every node v in the Patricia tree, let $b(v)$ be the binary string identifying it (i.e., in Figure, $b(k5) = 10101$). Let $|b(v)|$ be the length of $b(v)$. To maintain keys with arbitrary lengths, we also allow inner nodes with just one child in the Patricia trie, but only if such a node represents a key in the key set K (that is, node lists in the trie cannot be compressed beyond these nodes).

In our Patricia trie data structure, every node v stores $b(v)$ and the edge labels to the father and the two children, denoted by $p_{-}(v)$, $p_0(v)$ (the child whose edge label starts with 0), and $p_1(v)$ (the child whose edge label starts with 1), respectively. Node v also stores up to two keys, $key_1(v)$ and $key_2(v)$. If there is a key $k \in K$ with $k = b(v)$, then $key_1(v)$ is set to k and is otherwise empty. If there is a key $k \in K$ with prefix $b(v)$ that is longer than $b(v)$, then $key_2(v)$ is set to any one of these keys and is otherwise empty. As we will see, it is possible to make sure that every key $k \in K$ is stored at most once in $key_2(w)$ of some Patricia node w, which is important to ensure efficient updates. In order to remember this node, each node v with $b(v) \in K$ stores a string $r(v)$ with the property that for the node w with $b(w) = r(v)$, $key_2(w) = b(v)$.

It has already been shown that binary search is possible on a hashed trie [66], but to enable binary search on a hashed Patricia trie we need further nodes. For this we need some notation. Given two binary strings $a = (a_m, \cdots a_1, a_0)$ and $b = (b_m, \cdots b_1, b_0)$ (possibly filled up with leading 0s so that a and b have the same length), let $msd(a, b)$ be the unique bit position so that $a_j = b_j$ for all $j > msd(a, b)$ and $a_{msd(a,b)} \neq b_{msd(a,b)}$. So $msd(a, b)$ is the most significant bit in which the two strings differ. Now, let us

replace every edge $\{v, w\}$ in the Patricia trie by two edges $\{v, u\}$, $\{u, w\}$ where the label $b(u)$ of u is a prefix of $b(w)$. More precisely, let $\ell_1 = |b(v)|$ and $\ell_2 = |b(w)|$, and let $m = msd(\ell_1, \ell_2)$, then $b(u)$ is the prefix of $b(w)$ of length $\sum_{i=m}^{\lfloor \log \ell_2 \rfloor + 1} (\ell_2)_i \cdot 2^i$. (If u happens to have the same label as v or w, then we just merge it with that Patricia node and declare the result a Patricia node.)

As an example, assume we have two connected Patricia nodes v, w with $b(v) = 01001$ and $b(w) = 010010011$. Then $\ell_1 = |b(v)| = 5 = (101)_2$ and $\ell_2 = |b(w)| = 9 = (1001)_2$ and $m = msd(\ell_1, \ell_2) = 3$. The resulting msd-node u gets the prefix $b(u) = 01001001$ of length $8 = (1000)_2$. For an example see Figure 5.2.

The intuition for the introduction of the msd-nodes is to use the msd-nodes as nodes that will be visited by a binary search. Performing a binary search on a normal Patricia trie can lead to problems. Assume in round i a prefix k with length $|k| = l_i$ is searched, i.e. *HT-look-up(k)* is performed. If the Patricia trie and the hash table does not contain such a key, it is not clear whether the prefix length should be increased or decreased, as there could still be a key with a prefix longer than l_i. Therefore we introduce the msd-nodes v with the property that the length of $b(v)$ is calculated such that it will be found by the binary search.

We will call the resulting trie an *msd-Patricia trie* and the added nodes *msd-nodes* (while the original nodes are called *Patricia nodes*). In our data structure, each msd-node v stores $b(v)$ and the edge labels to the father and its child, using the variables $p_{-}(v)$ (for the father) and $p_0(v)$ or $p_1(v)$ (depending on whether the edge label to the child starts with 0 or 1). Note that the number of nodes in the *msd-Patricia Trie* is asymptotical the same as in the Patricia trie, i.e. we overload the Patricia trie only by a constant factor (one msd-node between a pair of connected Patricia nodes).

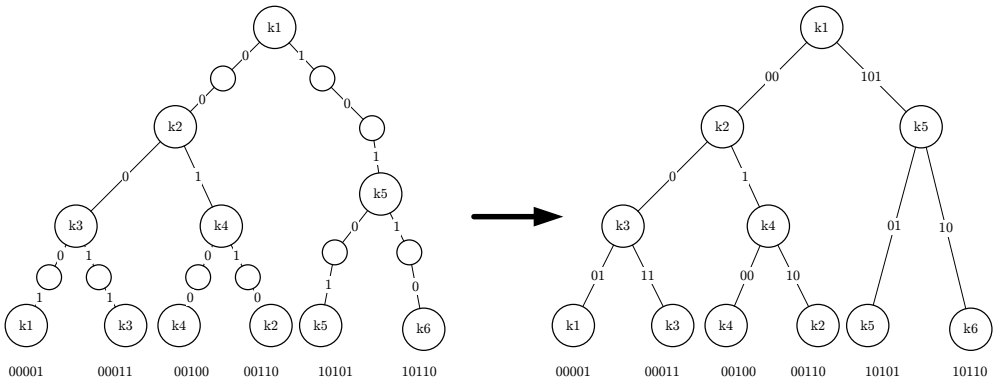


Figure 5.1: An example of a Patricia Trie

Hashed Patricia Trie

We will now show how to combine the msd-Patricia trie with a hash table. For every node (either a Patricia node or an msd-node) its label $b(v)$ will be hashed with an appropriate hash function (see Figure) for an example). At this moment we do not specify the hash function but just assume the existence of two operations on the used hash table: *HT-write(key, data)* and *HT-look-up(key)*, where $data$ represents the data of a Patricia node or msd-node with label key. Using these operations allows us to directly access nodes whose labels are known instead of traversing the Patricia trie. In order to keep our approach as general as possible, we will bound the cost of our operations in terms of hash table accesses and discuss later how to best realize the hash table in a distributed setting. We only focus on the communication costs

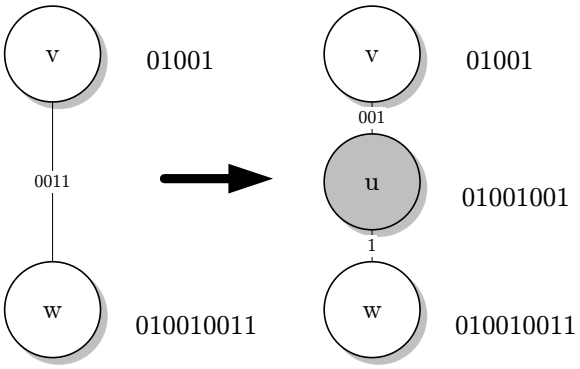


Figure 5.2: msd nodes are added between each pair of Patricia nodes

	HT-write(key,data)	*HT-look-up(key)*
PrefixSearch(x)	-	$\mathcal{O}(\log \|x\|)$
Insert(x)	$\mathcal{O}(1)$	$\mathcal{O}(\log \|x\|)$
Delete(x)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 5.1: Costs of hashed Patricia trie operations

since internal computations are considered to be rather cheap. More specifically, we present operations with a performance as shown in Table 5.1. For the memory usage we will show that the following theorem holds.

Theorem 5.2.1 *The hashed Patricia trie needs* $\Theta(\sum_{k\in K} |k|)$ *memory space, where* $\sum_{k\in K} |k|$ *is the sum of the bit lengths of the stored keys.*

Proof. Obviously it takes $\Omega(\sum_{k\in K} |k|)$ space to store all keys. But in a hashed Patricia trie not only the keys are stored, but also the inner nodes and the msd-nodes store their identifier $b(v)$ and the edge labels. We will show, that following our construction scheme the needed memory space is $\mathcal{O}(|K|)$. First, note that the Patricia trie is a binary tree with four kinds of nodes, the root r, which is an inner node with one or two children, the inner nodes V_2 with two children, the inner nodes V_1 with one children that store a $key_1(v) \in K$ and the leaves V_0 also storing keys in $key_1(v)$. Each node of V_2 has two descendants and each node of V_1 and the root one and every node except for the root has a parent, thus the total number of nodes is $2 * |V_2| + |V_1| + 2$, and the total number of nodes is also $1 + |V_2| + |V_1| + |V_0|$. Thus it holds that $|V_2| = |V_0| + 1$. Each $v \in V_2 \cup \{r\}$ stores a key in $key_2(v)$, each key is stored at most once in $key_2(v)$, and its identifier $b(v)$ satisfies $|b(v)| < |key_2(v)|$. Thus for all nodes of $V_2 \cup \{r\}$ $\mathcal{O}(\sum_{k\in K} |k|)$ space is required. For the msd-nodes we know that each msd-node is placed between every pair of connected Patricia nodes and each msd-node v has a child w with $|b(v)| < |b(w)|$. Thus the sum of all lengths of the $b(v)$s of the msd-nodes is less than the sum of all key lengths. At last note that the concatenation of the edge labels on a path from the root to a leaf corresponds to the key of the leaf, and therefore the sum of all edge labels must be less than the sum of the key lengths. Summing up $\Theta(\sum_{k\in K} |k|)$ memory space is needed.

□

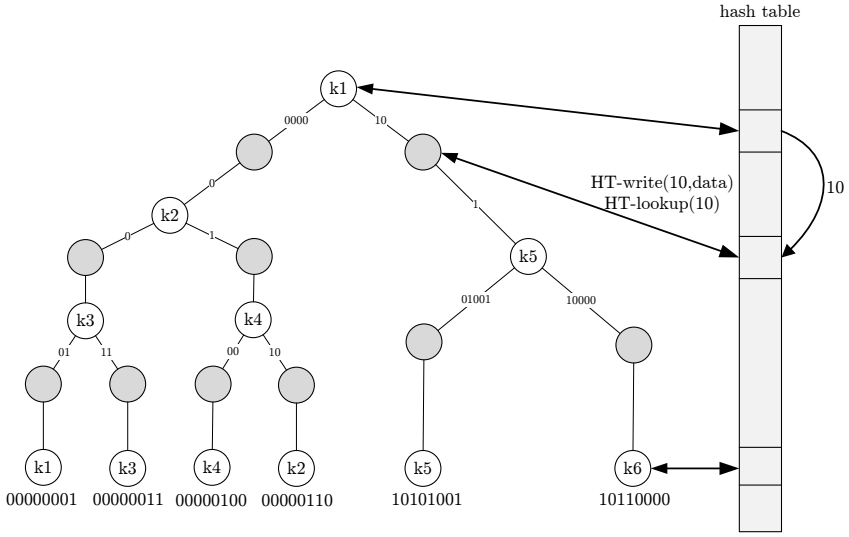


Figure 5.3: An example of a hashed Patricia trie

Applications

We can easily implement the hashed Patricia trie on top of a DHT. When using, e.g., Chord [75] or Pastry [70] in order to realize a DHT, then the *HT-write(key,data)* and *HT-look-up(key)* operations can be realized by sending a single message along at most $\mathcal{O}(\log N)$ sites w.h.p., where N is the number of sites in the system. Hence, we obtain the following result. By using an underlying DHT we leave the cases of churn and load balancing up to the DHT, i.e. our hashed Patricia trie does not affect the quality of the DHT.

Theorem 5.2.2 *When using Chord or Pastry the PrefixSearch(x) and the Insert(x) operations can be executed in $\mathcal{O}(\log |x| \log N)$ communication rounds w.h.p. The Delete(x) operation requires $\mathcal{O}(\log N)$ communication rounds w.h.p.*

5.2.4 Operations

In this section we provide algorithms for the operations *Insert(x)*, *Delete(x)* and *PrefixSearch(x)*. We will also show the correctness and bound their costs.

PrefixSearch(x)

The algorithm for the *PrefixSearch(x)* operation is based on the idea of binary searching. We assume that $x \in \{0,1\}^{\ell}$ has the form $(x_1, \ldots, x_\ell)$. The binary search starts with parameters $k = 0$ and $s = \lfloor \log(|x|) \rfloor$. Then in each binary search step the algorithm decides whether there is a key in the Patricia trie with a prefix $(x_1, \cdots x_{k+2^s})$. If so, then k is increased to $k + 2^s$, and otherwise k stays as it is. Also, s is decreased by 1 in any case. At last, the binary search provides some node v in the Patricia trie. From that node on, the Patricia trie is traversed downwards along the route determined by x till the first Patricia node w is found whose label is not a prefix of x, or a leaf is reached. *PrefixSearch(x)* then returns the key stored in w. As we will see, only a constant number of edges have to be traversed from v to w so that our

cost bound for *PrefixSearch(x)* holds. The pseudo code for this algorithm is given below. An illustration is given by Figure 5.4.

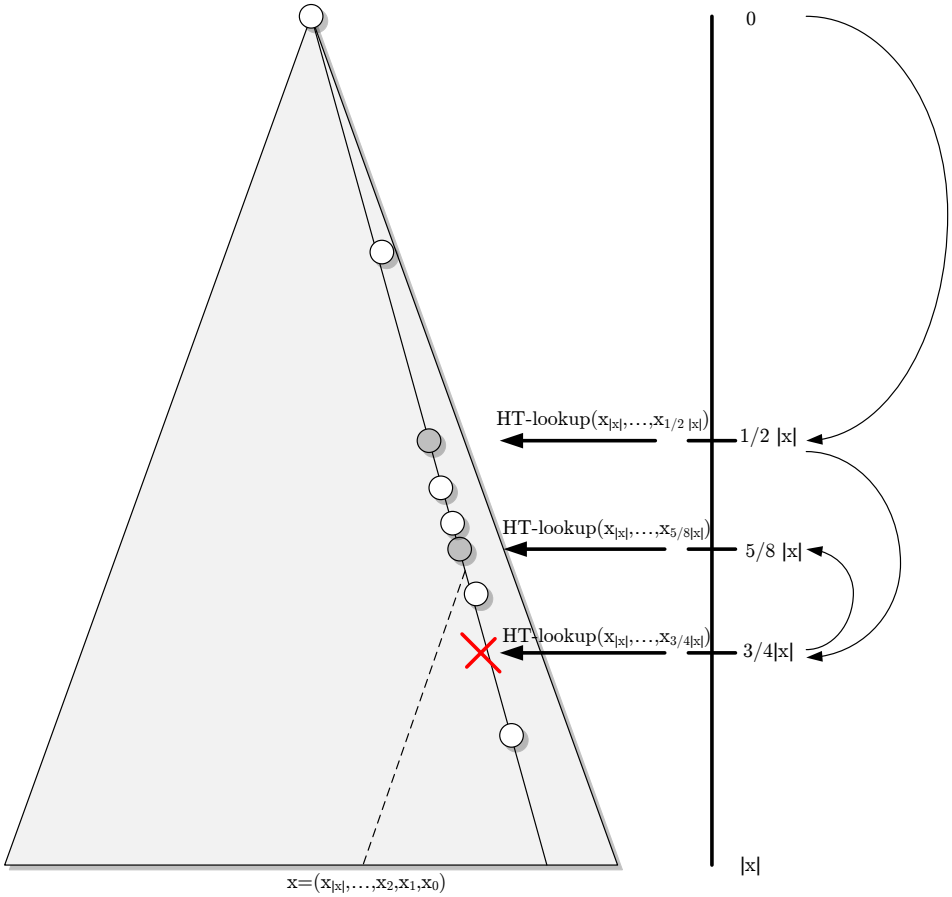


Figure 5.4: An illustration of a binary search on a hashed Patricia trie

We will prove the correctness of *PrefixSearch(x)* with the help of two lemmas. First, we will show the correctness of the first while loop (binary search) in *PrefixSearch(x)*. Let y be a key with the longest matching prefix pre_x with x in the data structure. Let v be the last node on the route from the root to pre_x in the msd-Patricia trie. Hence, $|b(v)| \leq |pre_x|$. As we said, $|b(v)| = l$ is represented by $(l_m, \cdots, l_0)$ differing from the representation of the keys. Thus $|b(v)|$ can also be described as $\sum_{j=0}^{m} 2^j * l_j$. We will now prove the following lemma.

Lemma 5.2.1 *If v is the last node on the unique path from the root to pre_x with an identifier $b(v)$ with $|b(v)| = \sum_{j=0}^{m} 2^j * l_j$ then in round k of the while loop in PrefixSearch(x) the existence of a prefix of x with length $\sum_{j=\lceil \log(|x|) \rceil - k+1}^{m} 2^j * l_j$ will be detected.*

Proof. The proof is given by induction on k. Assume that $b(v) \neq \epsilon$, i.e. v is not the root of the trie. Firstly we show that the theorem holds for the basis $k = 1$. Hence the binary search needs to detect a prefix of length $2^{\log(|x|)}$. Note that $\lceil \log(|x|) \rceil \geq m \geq 0$ as $|b(v)|$ has at most $\lceil \log(|x|) \rceil$ digits. Also note here is no prefix with length $2^{m'}$ and $m' > m$ as v is the last node on the route to pre_x. If $m < \log(|x|)$

Algorithm 5.2.1 PREFIXSEARCH(X)

if $key_1(HT - look - up(x)) = x$ or $key_2(HT - look - up(x)) = x$ **then**
  return x
  $s := \lfloor \log(|x|) \rfloor$
  $k := 0$
  $v := HT - look - up(\epsilon)$
  $p := p_{x_1}(v)$
while $s \geq 0$ **do**
  $v := HT - look - up(x_1, \cdots, x_{k+2^s})$
  if $v \neq \emptyset$ **then**
    $k := k + 2^s$
    $p := (x_1, \cdots, x_k) \circ p_{x_{k+1}}(v)$
  else
    if $(x_1, \cdots x_{k+2^s})$ is prefix of p **then**
      $k := k + 2^s$
  $s := s - 1$
while $b(v)$ is prefix of x **do**
  if $p_{x_{k+1}}(v)$ exists **then**
    $k := k + \left|p_{x_{k+1}}(v)\right|$
    $v := HT - look - up(b(v) \circ p_{x_{k+1}}(v))$
  else
    if $p_{\bar{x}_{k+1}}(v)$ exists **then**
      $k := k + \left|p_{\bar{x}_{k+1}}(v)\right|$
      $v := HT - look - up(b(v) \circ p_{\bar{x}_{k+1}}(v))$
    else
      break
if $key_1(v) \neq nil$ **then**
  return $key_1(v)$
else
  return $key_2(v)$

there is no prefix of length $2^{\log(|x|)}$, which corresponds to the undefined sum $\sum_{j=\lceil\log(|x|)\rceil-1+1}^{m} 2^j * l_j$. If $m = \lceil\log(|x|)\rceil$, then we claim that there exists a msd-node u with $b(u)$ being a prefix of $b(v)$ and $|b(u)| = 2^m \leq |b(v)|$ on the path to v. If v is a Patricia node, this msd-node exists because there must be a pair of original Patricia nodes u', v' on the route to v with $|b(u')| < 2^m$ and $|b(v')| \geq 2^m$ as $|b(w)|$ is increasing on the route to v beginning with 0 for the root. Then $msd(u', v') = m$. The same arguments hold if v is a msd-node and $|b(v)| > 2^m$. If v is a msd-node and $|b(v)| = 2^m$ $u = v$ and u' is the parent of v and v' its child.

In the induction step we show that, if the theorem holds for any $k < \lceil\log(|x|)\rceil$, it will also hold for $k+1$. By assuming that the theorem holds for k we know that a prefix of length $\sum_{j=\lceil\log(|x|)\rceil-k+1}^{m} 2^j * l_j$ was found by the binary search. If $\lceil\log(|x|)\rceil - k + 1 > m$ there is no prefix of this length and $\sum_{j=\lceil\log(|x|)\rceil-k+1}^{m} 2^j * l_j$ is empty and, so we assume from now on $\lceil\log(|x|)\rceil - k + 1 \leq m$. If $l_{k+1} = 0$ the already found prefix is also a prefix of length $\sum_{j=\lceil\log(|x|)\rceil-k+1}^{m} 2^j * l_j$. If $l_j = 1$ and there is a node u with $b(u)$ being a prefix of $b(v)$ and thus of pre_x and $|b(u)| = \sum_{j=\lceil\log(|x|)\rceil-k-1+1}^{m} 2^j * l_j$ it will be found by the binary search in round $k+1$. If $l_j = 1$ but there is no such node u with $|b(u)| = \sum_{j=\lceil\log(|x|)\rceil-k-1+1}^{m} 2^j * l_j$, then there cannot be a Patricia-node u' with $\sum_{j=\lceil\log(|x|)\rceil-k+1}^{m} 2^j * l_j \leq |b(u')| < \sum_{j=\lceil\log(|x|)\rceil-k-1+1}^{m} 2^j * l_j$, otherwise $l_{\lceil\log(|x|)\rceil-k-1}$ has to be a most significant digit for a pair of Patricia-nodes u', v', because a Patricia-node v' with $|b(v')| \geq \sum_{j=\lceil\log(|x|)\rceil-k-1+1}^{m} 2^j * l_j$ exists. The Patricia-node v' is either on the route to v or a child of v or v itself. We know by assumption that the binary search has found the last node w with $|b(w)| = \sum_{j=\lceil\log(|x|)\rceil-k'+1}^{m} 2^j * l_j$ $k' \leq k$, that exists in the msd-Patricia trie and set the variable p to $(x_1, \cdots x_{k'}) \circ p_{x_{k'+1}}$. Therefore $(x_1, \cdots x_{k+1})$ has to be a prefix of p and the prefix of length $\sum_{j=\lceil\log(|x|)\rceil-k-1+1}^{m} 2^j * l_j$ is detected by the binary search. This proves the theorem. If $b(v) = \epsilon$, i.e. $|b(v)| = 0$, then $m = 0$ and v will be detected as a prefix of x in the end of the binary search. □

Lemma 5.2.2 *If v is the last node on the route to pre_x, the next Patricia node v^* storing a key y that has the longest prefix pre_x with x will be found in a constant number of edge traverses.*

Proof. If v is a msd-node one edge will be traversed in the second while loop to reach v^*. If v is a Patricia-node there will be at most two edges traversed in the second loop. So after at most two steps v^* is reached following the edges to the descendants of v, that maximize the longest matching prefix. If v has no descendants $v^* = v$ and the loop breaks. □

Next, we analyze the cost of the operation in terms of *HT-write(key, data)* and *HT-look-up(key)* operations, which are asymptotically upper bounding the cost of all operations if we assume that all the other primitive operations, including computing the length of a string or concatenating two strings, can be done in unit time.

Theorem 5.2.3 *The PrefixSearch(x) operation requires $\mathcal{O}(\log(|x|))$ HT-look-up(key) operations.*

Proof. The first part of *PrefixSearch(x)* is the binary search that needs $O(\log(|x|))$ *HT-look-up(key)* operations operations. The second part of the *PrefixSearch(x)* needs at most two further *HT-look-up(key)* operations. So altogether the theorem follows. □

Insert(x)

The *Insert(x)* operation is similar to an insert in a traditional Patricia trie. This means: first, the correct position of x is searched in the trie and then x is inserted by redirecting the pointers of the affected

nodes. If the trie is empty we insert a new root w with $b(w) = \epsilon$ and $key_2(w) = x$ and one child v of w with $b(v) = x$ and $key_1(v) = x$ and $r(v) = b(w)$ and add an msd-node between w and v. Otherwise, we use a slightly adapted version of the *PrefixSearch(x)* operation, that returns the Patricia node v in the trie at which the operation stops instead of the key, to search for the right position in the trie. If $key_1(v) = x$ or $key_2(v) = x$, the key is already in the data structure and will not be inserted a second time. If we know v we also know u as the parental Patricia node via the $p_{\_}(v)$ edges. If x is a prefix of $b(v)$ there can be three cases. If $x = b(u)$ then $key_1(u) = x$. If $x = b(v)$ then $key_1(v) = x$. And if $|b(u)| < |x| < |b(v)|$ then a node w is inserted between u, v with $b(w) = x$ and $key_1(w) = x$ and the msd-nodes between u, w and w, v are either added or their edges are updated (the old msd-node remains either between u and w or w and v). If $b(v)$ is a prefix of x, then v has to be a leaf and a node w with $key_1(w) = x$ and $b(w) = x$ is inserted as a child of v. Further $r(w)$ is set to $r(v)$ and $r(v)$ is deleted and $key_2(HT - look - up((r(v)))) = x$ is set. Finally the msd-node between v, w is inserted. Recall that $r(v)$ returns the inner node storing $key_2(HT - look - up(x)) = x$. If neither x is a prefix of $b(v)$ nor $b(v)$ a prefix of x then a new Patricia node w with $key_1(w) = x$ and $b(w) = x$ is inserted as a sibling of v. If $p_{x_{|b(u)|+1}}$ exists, a Patricia node w' has to be inserted between u, v as the new parent of v and w. Let $j = msd(b(v), x)$, then $key_2(w') = x$ and $b(w') = x_1 \cdots x_{j-1}$. Then $r(w) = b(w')$ and the msd-nodes between u, w' w', v and w', w are added or updated. If $p_{x_{|b(u)|+1}}$ does not exist a node w with $key_1(w) = x$ and $b(w) = x$ is inserted as a child of u. If $key_2(u) = nil$ it is updated to $key_2(u) = x$ and $r(w) = b(u)$. At last the msd-node between u, w is added.

Obvious *Insert(x)* is correct if *PrefixSearch(x)* is correct. E.g. in Figure 5.5.

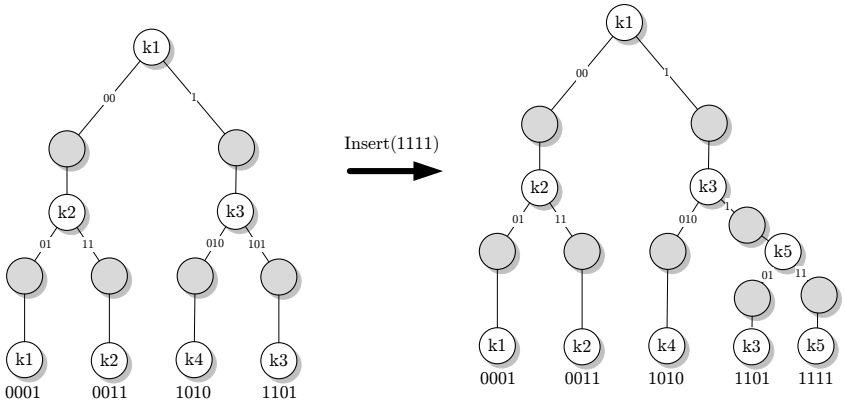


Figure 5.5: How to insert a new key into a hashed Patricia trie

Theorem 5.2.4 *The Insert(x) operation requires $\mathcal{O}(\log(|x|))$ HT-look-up(key) operations(key) operations and $\mathcal{O}(1)$ HT-write(key,data) operations.*

Proof. The *Insert(x)* operation needs one *PrefixSearch(x)* with $\mathcal{O}(\log(|x|))$ *HT-look-up(key)* operations, and afterward a constant number of nodes are inserted by *HT-write(key,data)* operations. So *Insert(x)* can be executed with $\mathcal{O}(\log(|x|))$ *HT-look-up(key)* operations and $\mathcal{O}(1)$ *HT-write(key,data)* operations. □

Delete(x)

We assume that the key x exists. Then we hash the key to get the corresponding nodes $v = HT - look - up(x)$, $w = HT - look - up(r(v))$ and u as v's parental Patricia node. If $key_2(v) \neq nil$, v

is still needed after the deletion of x, so only $key_1(v) = nil$, has to be set, as there is no node w. If $key_2(v) = nil$, v can have one or no child. If there is a child w' it is connected to u, and a new msd-node between u and w' is inserted after the deletion of the msd-nodes caused by v.

If v has no child, v is a leaf. In this case if $key_2(u) = nil$, so v was the only child of u, $r(u) = r(v)$ and $key_2(w) = key_1(u)$. If $key_2(u) \neq nil$, the data structure is updated by $key_2(w) = key_2(u)$ and $r(HT - look - up(key_2(u))) = b(w)$ and $key_2(u) = nil$, as u is no longer necessary for the structure of the Patricia trie. If $key_1(u) \neq nil$, u is still in the data structure due to the key $key_1(u) = b(u)$. Otherwise u can be deleted by connecting the parental Patricia node and the remaining child of u. Then the msd-nodes caused by u are deleted and new msd-nodes between the parental Patricia node and the child are inserted.

Obviously *Delete(x)* only substitutes nodes with one descendant by an edge and therefore maintains the msd-Patricia trie structure correctly. E.g. in Figure 5.6.

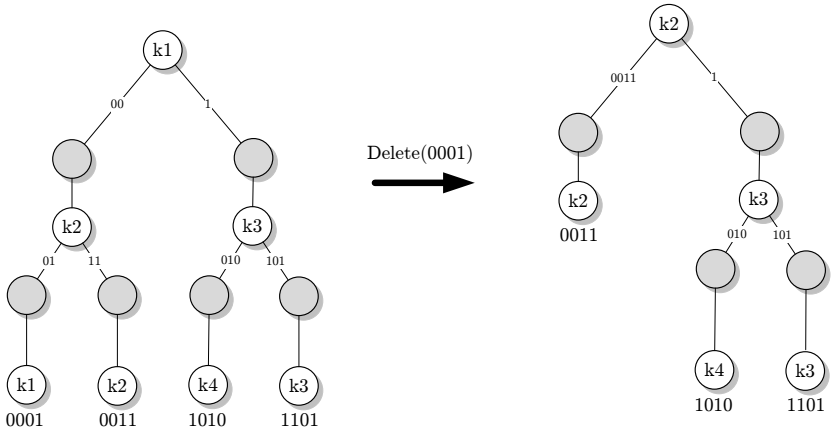


Figure 5.6: How to delete a key in a hashed Patricia trie

Theorem 5.2.5 *The Delete(x) operation requires $\mathcal{O}(1)$ HT-look-up(key) operations and HT-write(key,data) operations.*

Proof. The *Delete(x)* operation needs a constant number of *HT-look-up(key)* and *HT-write(key,data)* operations, as only a constant number of Patricia and msd-nodes are modified or deleted. □

5.3 Hashed Predecessor Patricia Trie: Efficient Predecessor Queries on Distributed Hash Tables

5.3.1 Introduction

In this section we extend the hashed Patricia Trie to be able to support further queries. In particular we address the problem of finding the predecessor in a key set and present an efficient data structure called hashed Predecessor Patricia trie. Our hashed Predecessor Patricia trie supports *PredecessorSearch(x)* and *Insert(x)* and *Delete(x)* in $\mathcal{O}(\log \log U)$ hash table accesses when U is the size of the universe of the keys. That is the costs only depend on U and not the size of the data structure. One feature of our approach is

that it only uses the look-up interface of the hash table and therefore hash table accesses may be realized by any distributed hash table (DHT). In the following we define the predecessor problem and present a data structure that efficiently supports the predecessor problem with the help of any common DHT, e.g. Chord [75] or Pastry [70].

Definition 5.3.1 (Predecessor Problem) *Given a key set S with a total order and a search key x, find* $\max\{y \in S | y \leq x\}$.

In the context of strings, a natural interpretation of $z \leq x$ is that z is lexicographically smaller than x (i.e., the first letter different in x and z is lexicographically smaller in z than in x). Throughout the paper we will only consider binary strings.

A data structure for the predecessor problem supports the following operations:

- *Insert(x)*: this adds the key x to the set S. If x already exists in S, it will not be inserted a second time.
- *Delete(x)*: this removes the key from the set S, i.e., $S := S - \{x\}$.
- *PredecessorSearch(x)*: this returns a key $y \in S$ that is the predecessor of x.

We will present a data structure called *hashed Predecessor Patricia trie* for these operations, which is based on an extension of the Patricia trie [56] called *hashed Patricia trie*. For the hashed Predecessor Patricia trie we will show that all operations *PredecessorSearch(x)*, *Insert(x)* and *Delete(x)* can be executed in $\mathcal{O}(\log\log U)$ hash table accesses when U is the size of the universe of the keys. That is, the costs of the presented operations are independent of the size of the data structure as they only depend on the size of the universe U. Our bounds on the hash table accesses imply that only a few messages have to be sent when using the well-known distributed hash table (DHT) approach in order to realize our data structure. We further point out that our solution is asymptotically optimal in the use of memory space, requiring only $\Theta(\text{sum of all key lengths})$ of memory. This implies that the data structure can easily be implemented on top of a DHT without producing an overhead in terms of memory usage.

5.3.2 Our contributions

We present a data structure that supports predecessor queries on distributed hash tables and give the costs of the operations *PredecessorSearch(x)*, *Insert(x)*, and *Delete(x)*, given in theorems 5.3.2- 5.3.4. With these results we elaborate on the results mentioned in [41]:

S. Kniesburges, C. Scheideler, Brief Announcement: Hashed Predecessor Patricia Trie - A Data Structure for Efficient Predecessor Queries in Peer-to-Peer Systems, 26th International Symposium on Distributed Computing (DISC), 2012.

5.3.3 Hashed Predecessor Patricia trie

In this section we describe our solution. The hashed Predecessor Patricia trie is based on the hashed Patricia trie we introduced in the last section. As for the hashed Patricia trie we do not specify the underlying hash function but just assume the existence of two operations on the used hash table: *HT-write(key, data)* and *HT-look-up(key)*, where $data$ represents the data of a Patricia node or msd-node with

label key. Using these operations allows us to directly access nodes whose labels are known instead of traversing the Patricia trie. In order to keep our approach as general as possible, we will bound the cost of our operations in terms of hash table accesses.

The binary search in the hashed Patricia trie consists of two parts as described in the last section. The first part returns the node in the hashed Patricia trie of which the identifier is the longest prefix of the search key among all nodes in the hashed Patricia trie. In the second part a key is returned that has the longest prefix with the search key among all keys.

We will reuse the first part of the binary search to find the predecessor of a search key. If we have found the node v of which the label is the longest prefix of the search key x, then there are 3 possible cases of the position of the predecessor $pred(x)$ of the search key:

- x is between the left and right sub-trie, so $pred(x)$ is the greatest key in the left sub-trie.
- x is greater then all keys in the right sub-trie, then $pred(x)$ is the greatest key in the left sub-trie of the first ancestor of v containing v in its left sub-trie.
- x is smaller then all keys in the left sub-trie, then $pred(x)$ is the greatest key in the left sub-trie of the first ancestor of v containing v in its right sub-trie.

By adding a pointer l_{max} to the greatest key in the left sub-trie of each Patricia node we can easily solve two of the three cases. Following this pointer from v we can always solve the first case, following the l_{max} pointer from the parental Patricia node $parent(v)$ of v we can either solve the second or the third case as v is either in the left or right sub-trie of $parent(v)$. Thus the remaining case to solve is the case in which $pred(x)$ is greater (resp. smaller) then all nodes in the sub-trie rooted at v and v is not in the left (resp. right) sub-trie of $parent(v)$. An intuitive approach would be to add pointers to the maximal and minimal key in the sub-trie rooted at v. As many nodes can point to the same maximal key, e.g. v is the right child of its parental Patricia node, which again is the right child of its parental node and so on, this approach would lead to high update costs of $\Theta(\log U)$. In our solution we will only store one pointer for each leaf as a maximal (resp. minimal) key in a sub-trie. The first node v on the unique path from the root to the leaf for which a leaf is the maximal (resp. minimal) key in its sub-trie stores a pointer r_{max} (resp. l_{min}) to it. For all other Patricia nodes, that have the same r_{max} (resp. l_{min}) as v we add a pointer structure such that they will find v and thus r_{max} in at most $\mathcal{O}(\log \log U)$ steps. To describe this pointer structure we introduce the following definitions.

Definition 5.3.2 *We call a path along edges, of which each label starts with a* 1 *(resp.* 0*) and that ends in a leaf node of the trie a* 1-sequence *(resp. a* 0-sequence*). Each such 1-sequence (0-sequence) is defined by its starting node v in the trie, i.e. the last node on the path to the leaf such that $p_{-}(v) = 0\cdots$ (resp. $p_{-}(v) = 1\cdots$).*

Definition 5.3.3 *Let w be a msd-node with $|b(w)| = \sum_{i=0}^{\log \log U} x_i 2^i$ and let l be the last digit such that $x_l = 1$ and $x_j = 0 \forall j < l$. Then w' is the msd-node that is visited before w such that $b(w')$ is the longest prefix of $b(w)$ among all prefixes of lengths $|b(w')| = \sum_{i=j}^{\log \log U} x_i 2^i$ $j > l$. We call w a* first hit *of the binary search if w and w' lie on different 1-sequences (resp. 0-sequences).*

For our pointer structure we determine at which msd-nodes a binary search can firstly hit a 1-sequence (resp. 0-sequence). All nodes along such a sequence have the same r_{max} (resp. l_{min}) leaf. The determined msd-nodes will then be connected as a linked list by pointers $ll(v)$. Thus if we find a node v in the binary

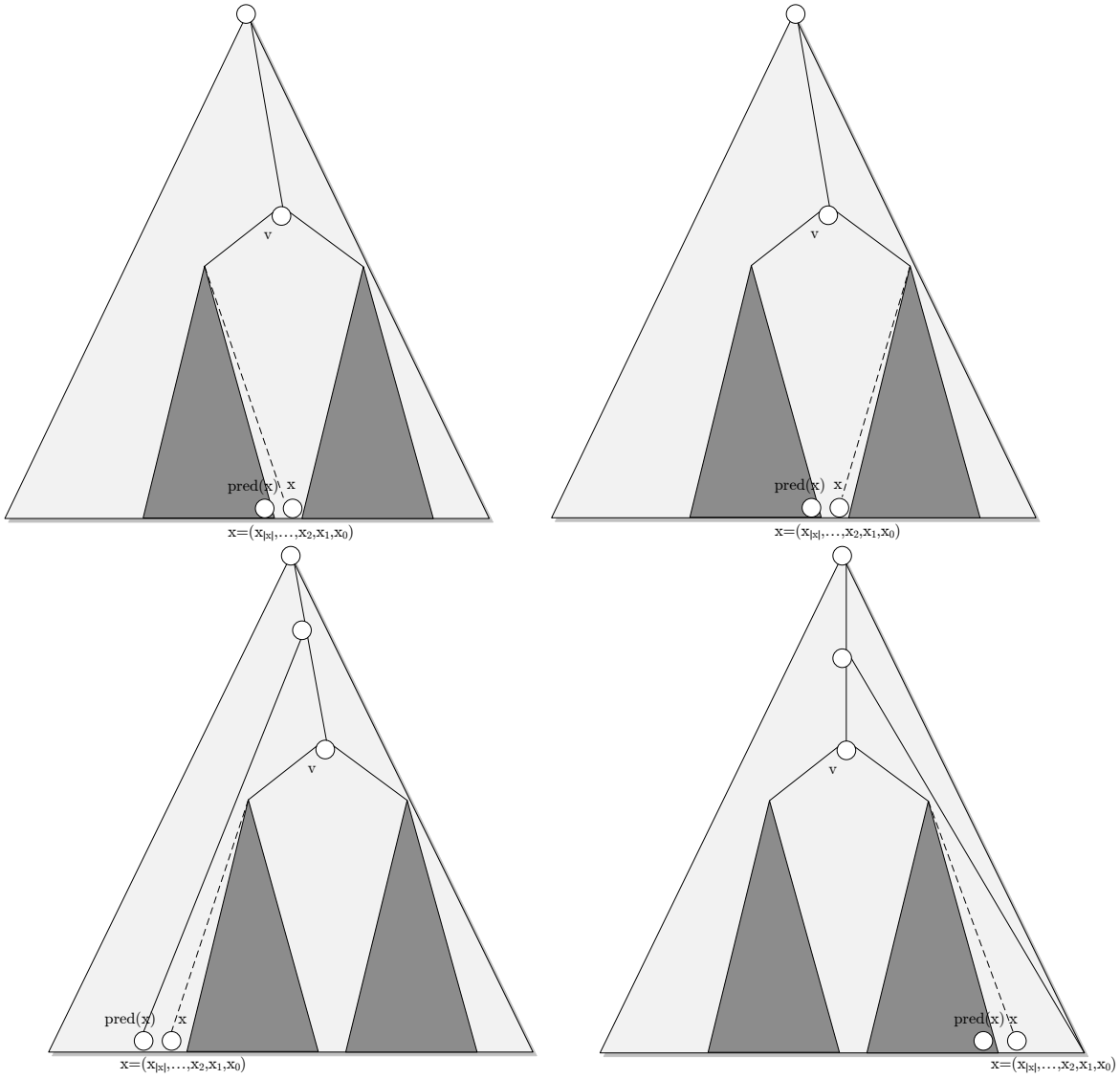


Figure 5.7: How to find $pred(x)$ from v

search, we follow the linked list pointers to find the starting node of that sequence, which stores the corresponding r_{max} (resp. l_{min}) value.

Lemma 5.3.1 *A linked list by pointers $ll(v)$ consists of at most $\mathcal{O}(\log\log U)$ nodes, i.e. there are at most $\mathcal{O}(\log\log U)$ first hits on a 1-sequence (resp. 0-sequence).*

Proof. Let u be a starting node of a 1-sequence (resp. 0-sequence) and let w be the corresponding r_{max} leaf. Then we can determine all first hits on this sequence. For each binary search for a key x such that $b(u)$ is a prefix of x, i.e. a binary search that will actually hit the 1-sequence, the first possible hit, i.e. the first prefix length of x that is queried, is given by the msd-node v_1 with $|b(v_1)| = \sum_{i=m}^{\lfloor \log \ell_2 \rfloor + 1} (\ell_2)_i \cdot 2^i$, where $\ell_1 = |b(u)|$, $\ell_2 = |b(w)|$ and $m_1 = msd(\ell_1, \ell_2)$ and $b(v_1)$ is a prefix of w. Note that all nodes v' on the 1-sequence starting by u with $|b(v')| > |b(v_1)|$ can not be first hits. If we assume that such a node v' is a first hit, then $b(v')$ is a prefix of the search key x, but then also $b(v_1)$ is a prefix of x and v_1 is queried before v', such that v_1 is a first hit, which contradicts the assumption that v' is a first hit. Then for another first hit v_2 on the same 1-sequence it must hold that $|b(v_2)| < |b(v_1)|$. This meas that $b(v_1)$ is no prefix of the search key x and the query of the binary search was negative. Then the next possible hit or the prefix length of the next query by the binary search is given by v_2 with $|b(v_2)| = \sum_{i=m}^{\lfloor \log \ell_2 \rfloor + 1} (\ell_2)_i \cdot 2^i$, where $\ell_1 = |b(u)|$, $\ell_2 = |b(v_1)|$ and $m_2 = msd(\ell_1, \ell_2)$ and $b(v_2)$ is a prefix of w. Note that all nodes v'' on the 1-sequence starting by u with $|b(v_1)| > |b(v'')| > |b(v_2)|$ can not be first hits by the same arguments as for the nodes v'. By induction we can show that the j-th first hit is given by v_j with $|b(v_j)| = \sum_{i=m}^{\lfloor \log \ell_2 \rfloor + 1} (\ell_2)_i \cdot 2^i$, where $\ell_1 = |b(u)|$, $\ell_2 = |b(v_{j-1})|$ and $m_j = msd(\ell_1, \ell_2)$ and $b(v_j)$ is a prefix of w. Then j is limited by $\mathcal{O}(\log\log U)$ as for each first hit v_j we determined a different msd-value m_j and the number of bits is limited by $\mathcal{O}(\log\log U)$. Or in other words to reach v_j there have to be $j-1$ negative queries by the binary search and according to Theorem 5.2.3 in the last section the number of queries (*HT-look-up(key)* operations) of the binary search is limited by $\mathcal{O}(\log\log U)$. □

An example of the added pointers is given in Fig. 5.8. For the memory usage we will show that the following theorem holds.

Theorem 5.3.1 *The hashed Predecessor Patricia trie needs $\Theta(\sum_{k\in S} |k|)$ memory space, where $\sum_{k\in S} |k|$ is the sum of the bit lengths of the stored keys.*

Proof. According to Theorem 5.2.1 a hashed Patricia trie needs $\Theta(\sum_{k\in S} |k|)$ memory space. Additionally each node stores at most one l_{max} and r_{max} pointer and one further pointer to build a linked list along a 1-sequence or a 0-sequence. Thus the needed memory space increases only by a constant factor and stays asymptotically optimal. □

5.3.4 Operations

PredecessorSearch(x)

The algorithm for the *PredecessorSearch(x)* operation is based on the idea of binary searching presented in the last section and [40]. We assume that $x \in \{0,1\}^\ell$ has the form $(x_1, \ldots, x_\ell)$. The binary search starts with parameters $k = 0$ and $s = \lfloor \log(|x|) \rfloor$. Then in each binary search step the algorithm decides whether there is a key in the Patricia trie with a prefix $(x_1, \cdots x_{k+2^s})$. If so, then k is increased to $k + 2^s$,

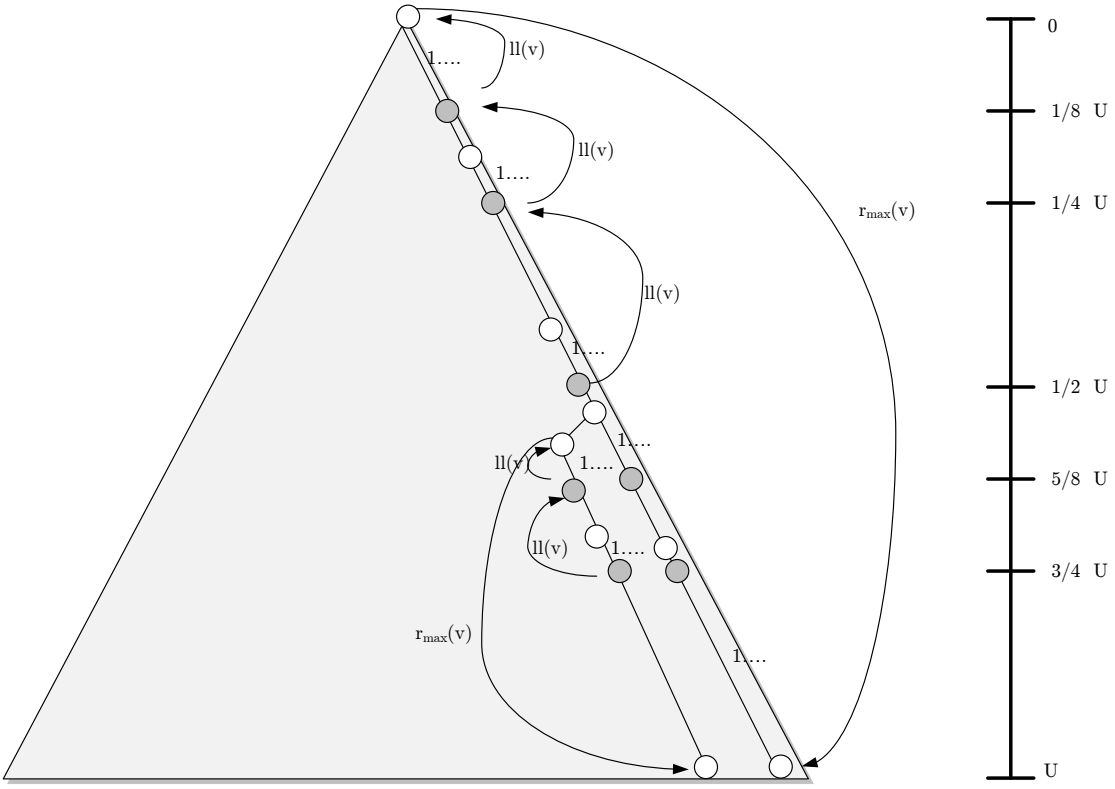


Figure 5.8: Two 1-sequences with linked lists and r_{max} pointers

and otherwise k stays as it is. Also, s is decreased by 1 in any case. At last, the binary search provides some node v in the Patricia trie of which the identifier is a longest prefix of the search key x. Along this binary search the $ll(v)$ value is updated and when the binary search terminates it follows the $ll(v)$ pointer to reach the correct node that stores the r_{max} (resp. l_min) value for v.

Algorithm 5.3.1 PredecessorSearch(x)

if $key_1(HT - look - up(x)) = x$ or $key_2(HT - look - up(x)) = x$ **then**
  return x
else
  $s := \lfloor \log(|x|) \rfloor$
  $k := 0$
  $v := HT - look - up(\epsilon)$
  $ll := ll(v)$
  $p := p_{x_1}(v)$
while $s \geq 0$ **do**
  $v := HT - look - up(x_1, \cdots, x_{k+2^s})$
  if $v \neq \emptyset$ **then**
    $k := k + 2^s$
    $p := (x_1, \cdots, x_k) \circ p_{x_{k+1}}(v)$
    if $ll(v) \neq \emptyset$ **then**
      $ll := ll(v)$
  else
    if $(x_1, \cdots x_{k+2^s})$ is prefix of p **then**
      $k := k + 2^s$
  $s := s - 1$
$w := HT - look - up(b(v) - p_-(v))$
if $l_{max}(v) < x < l_{max}(w)$ **then**
  return $l_{max}(v)$
else if $l_{max}(w) < x < l_{max}(v)$ **then**
  return $l_{max}(w)$
else
  while $ll \neq null$ **do**
    $ll := ll(ll)$
return $l_{min}(ll)$ or $r_{max}(ll)$

We now show the correctness and complexity of the predecessor search.

Theorem 5.3.2 *An execution of PredecessorSearch(x) needs $\mathcal{O}(\log \log U)$ HT-look-up(key) operations.*

Proof. In the last section and [40] we have shown that a binary search on the prefix length is possible on a hashed Patricia trie in $\mathcal{O}(\log \log U)$ steps. This binary search finds the node v, such that v's string identifier $b(v)$ is the longest prefix for the query key x for all string identifiers of nodes in the trie. To find the correct predecessor of x at most $\mathcal{O}(\log \log U)$ further steps along $ll(v)$ pointers are necessary. □

Insert(x)

The *Insert(x)* operation is similar to an insert in a traditional Patricia trie. This means: first, the correct position of x is searched in the trie and then x is inserted by redirecting the pointers of the affected nodes. To insert a key x we use the binary search to find the Patricia node v such that v's string identifier $b(v)$ is the longest prefix for the query key x for all Patricia nodes and as in the Predecessor Search we also find the node v' that is the first hit of the binary search before finding v. Following $ll(v')$ we can also determine the node u that stores the corresponding r_{max} (resp. l_min), i.e. u defines the 1-sequence (resp. 0-sequence). If the binary search ends in a msd-node v is simply its parental Patricia node. This binary search takes at most $\mathcal{O}(\log\log U)$ steps. If $key(v) = x$, the key is already in the data structure and will not be inserted a second time. Otherwise if we follow $p_z(v)$ with $b(v) \circ z$ being a prefix of x to the next Patricia node w. Then between w and v a new Patricia node w' is added, such that $b(w')$ is the longest common prefix of x and $b(w)$. Now there can be four cases:

- $z = 1$ and x is stored in a left child of w'
- $z = 1$ and x is stored in a right child of w'
- $z = 0$ and x is stored in a right child of w'
- $z = 0$ and x is stored in a left child of w'

We only consider the first two cases as the later ones can be solved analogously. In the first case we insert w' as a right child of v between v and w. Additionally we insert three msd-nodes m_0, m_1 and m_2 between v and w', w' and x and w' and w. As m_1 is the only msd-node on the 0-sequence from w' to x we set $ll(m_1) = w'$. Note that for the 1-sequence defined by u along v, w' and w all msd-nodes that were first hits of the binary search remain first hits and we only have to include m_0 or m_2 in the linked list of first hits, if they are also first hits. This can easily be done as we know the last first hit v' on the path to v. The second case is more complicated as it might happen that we create a new 1-sequence defined by w and thus msd-nodes that were no first hits on the 1-sequence defined by u now become first hits (see figure). In this case we again add w' as a right child of v and w becomes a left child of w'. Additionally we insert three msd-nodes m_0, m_1 and m_2 between v and w', w' and w and w' and x. As m_2 is the only msd-node on the 1-sequence from w' to x we set $ll(m_2) = v'$ and add it to the linked list if m_2 is a first hit. For the new 1-sequence defined by w we determine all possible first hits. Let $|b(w)| = \sum_{i=0}^{\log\log U} x_i 2^i$. Then let $l = msd(\log\log U, |b(w)|)$. Then each first hit between w and the old r_max has to be a msd-node with an identifier that is a prefix of r_max of length $\sum^{\log\log U} i = lx_i 2^i + 2^l$ $\forall l : x_l = 0$. Additionally we have to set $l_min(w') = l_min(w)$, as w is now a left child of w' and thus the 0-sequence starts at w'. We update the first hits of this 0-sequence in the same way as described for the 1-sequence. See Figure 5.9.

Theorem 5.3.3 *An execution of Insert(x) needs $\mathcal{O}(\log\log U)$ HT-look-up(key) and HT-write(key,data) operations.*

Proof. The correctness follows from the correctness of *PredecessorSearch(x)* and the given description. According to 5.3.2 *PredecessorSearch(x)* needs at most $\mathcal{O}(\log\log U)$ *HT-look-up(key)* operations. We insert a constant number of new nodes into the hashed Patricia trie and have to update the $ll(v)$ pointer of msd-nodes becoming first hits. It follows from the given description that there are at most $\mathcal{O}(\log\log U)$ of such msd-nodes. Thus *Insert(x)* needs at most $\mathcal{O}(\log\log U)$ *HT-write(key,data)* operations. □

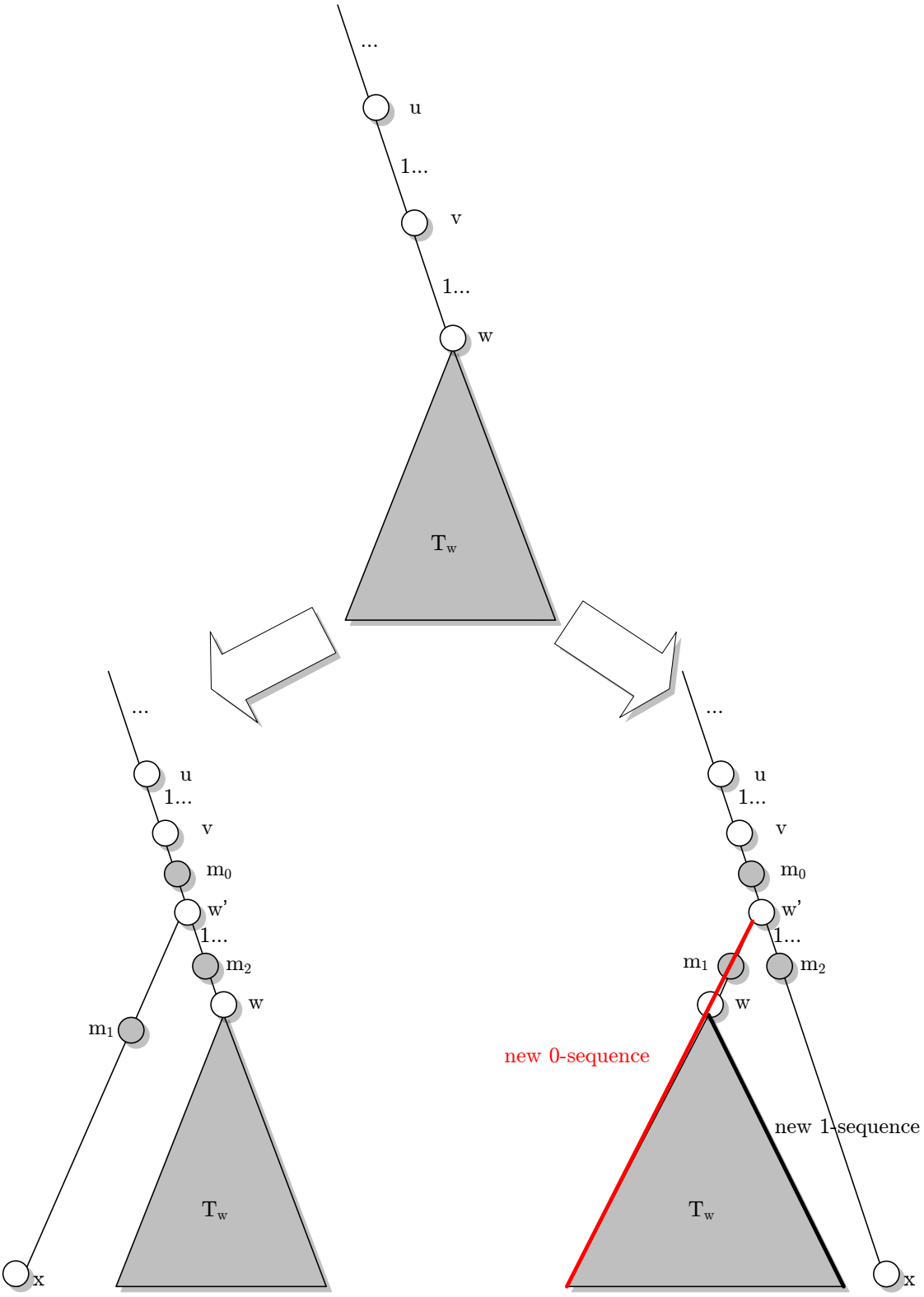


Figure 5.9: Illustration of *Insert(x)*

Delete(x)

The *Delete(x)* operation is performed as a reverse Insert(x) operation, i.e. first x is hashed to get access to the leaf node with the label x. Then this node and its parental Patricia node w' have to be removed and the pointers in the hashed Patricia trie have to be updated, as x is not longer r_max (resp. l_min) for a 1-sequence (resp. 0-sequence). Furthermore the parental Patricia node w' has defined a 1-sequence or 0-sequence and thus the $ll(v)$ pointers have to be updated. Like in the *Insert(x)* operation we also find the node u defining the 1-sequence w' and its parental Patricia node v are lying on by the search in the *PredecessorSearch(x)* operation. Using the same notation as for *Insert(x)*, again there can be four cases:

- $z = 1$ and x is stored in a left child of w'
- $z = 1$ and x is stored in a right child of w'
- $z = 0$ and x is stored in a right child of w'
- $z = 0$ and x is stored in a left child of w'

We again only consider the first two cases as the later ones can be solved analogously. In the first case we remove w' as a right child of v and connect v to the right child of w' w. We also remove the msd-nodes m_0 and m_2 between v and w' and w and w'. As $|b(w')| > |b(w)|$ one msd-node out of m_0 and m_2 is kept and thus we do not have to update the linked list of first hits for the 1-sequence. As the 0-sequence defined by w' only consists of w' m_1 and the removed leaf, we do not have to perform further updates. Like in the Insert(x) operation the second case is more complicated. In this case w' is left child of w and defines a new 1-sequence with another linked-list. Then the linked lists of the 1-sequences defined by u and w' have to be merged. Furthermore w now defines the 0-sequence that was defined by w'. Thus we connect w as a right child to v. As we know r_{max} of the 1-sequence defined by w we can update the r_{max} value stored in u and can also update the linked-list by determining all possible first hit on the 1-sequence defined by u. Let $|b(u)| = \sum_{i=0}^{\log\log U} x_i 2^i$. Then let $l = msd(\log\log U, |b(u)|)$. Then each first hit between u and the new r_max has to be a msd-node with an identifier that is a prefix of r_max of length $\sum^{\log\log U} i = lx_i 2^i + 2^l \; \forall l : x_l = 0$. We also set l_{min} stored at w to the l_{min} value previously stored at w' and update the linked list as described above for the 1-sequence. See Figure 5.10.

Theorem 5.3.4 *An execution of Delete(x) needs $\mathcal{O}(\log\log U)$ HT-look-up(key) and HT-write(key,data) operations.*

Proof. The correctness follows from the correctness of *PredecessorSearch(x)* and the given description. According to 5.3.2 *PredecessorSearch(x)* needs at most $\mathcal{O}(\log\log U)$ *HT-look-up(key)* operations. We insert a constant number of new nodes into the hashed Patricia trie and have to update the $ll(v)$ pointer of msd-nodes becoming first hits. It follows from the given description that there are at most $\mathcal{O}(\log\log U)$ of such msd-nodes. Thus *Delete(x)* needs at most $\mathcal{O}(\log\log U)$ *HT-write(key,data)* operations. □

5.4 Conclusion & Outlook

In this chapter we presented two simple data structures that can be built on top of existing distributed hash tables to support longest prefix queries and predecessor queries or range queries. The hashed Patricia trie we introduced in the first section supports *PrefixSearch(x)* and *Insert(x)* with $\mathcal{O}(\log(|x|))$ hash table

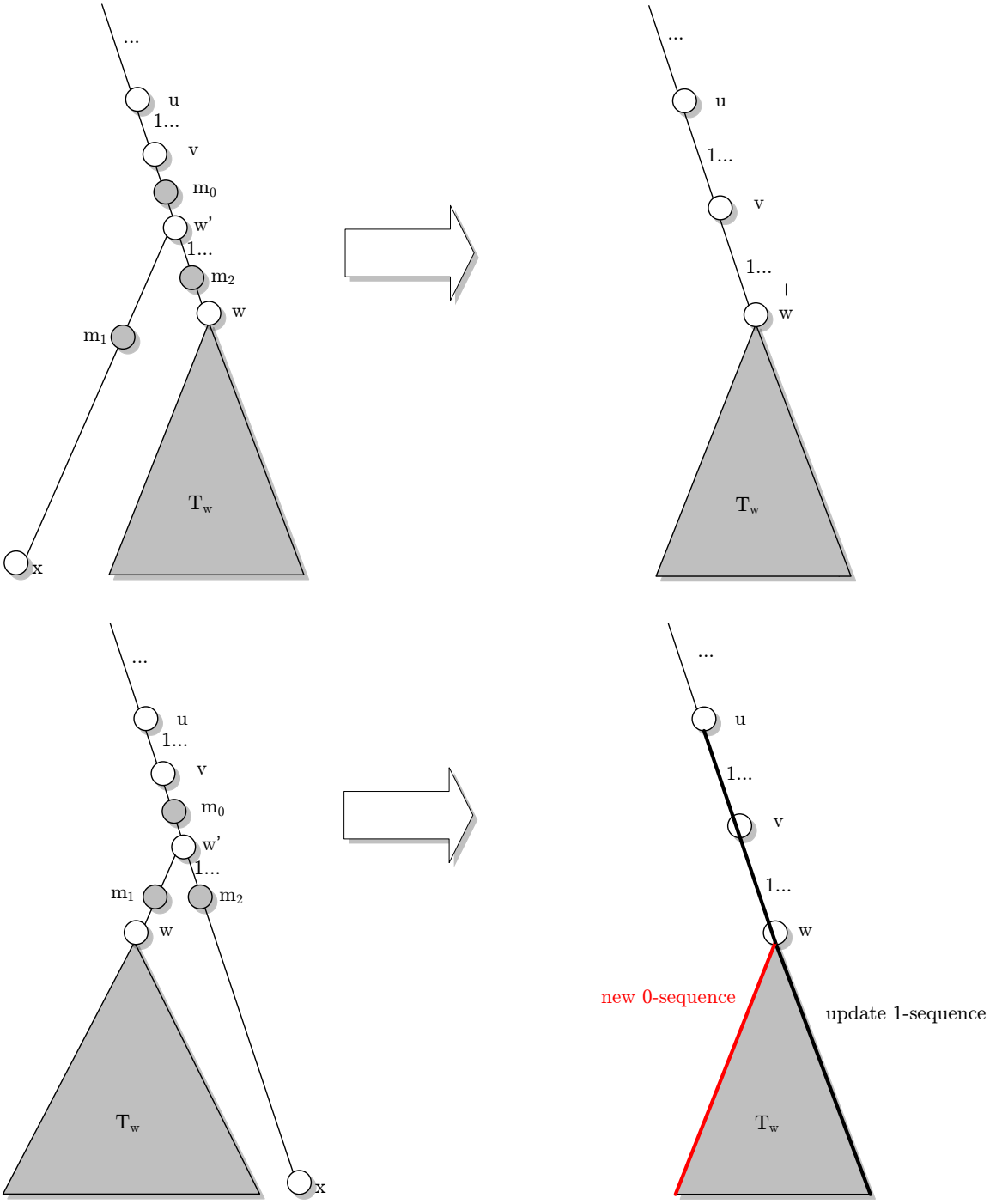


Figure 5.10: Illustration of *Delete(x)*

accesses while *Delete(x)* only needs a constant number of hash table accesses. The hashed predecessor Patricia trie introduced in the second section supports all operations *PredecessorSearch(x)*, *Insert(x)* and *Delete(x)* using only $\mathcal{O}(\log \log U)$ hash table accesses when U is the size of the universe of the keys. As we have already mentioned the runtime in terms of hash table accesses is optimal for both approaches as was shown in [63]. It remains an open question if the number of update operations, i.e. *HT-write(key,data)* operations can be reduced when supporting predecessor queries. Although there are several other approaches to support range queries on distributed hash tables we believe that our suggested solutions are more intuitive and simple to understand.

List of Figures

List of Algorithms

Bibliography

[1] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 335–342. ACM, 2000.

[2] James Aspnes and Gauri Shah. Skip graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2003, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 384–393. ACM/SIAM, 2003.

[3] Baruch Awerbuch and Christian Scheideler. Peer-to-peer systems for prefix search. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 123–132. ACM, 2003.

[4] Roberto Baldoni, Marin Bertier, Michel Raynal, and Sara Tucci Piergiovanni. Looking for a definition of dynamic distributed systems. In Victor E. Malyshkin, editor, *Parallel Computing Technologies, 9th International Conference, PaCT 2007, Pereslavl-Zalessky, Russia, September 3-7, 2007, Proceedings*, volume 4671 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2007.

[5] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[6] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[7] Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2011.

[8] Marcin Bienkowski, André Brinkmann, Marek Klonowski, and Miroslaw Korzeniowski. Skewccc+: A heterogeneous distributed hash table. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2010.

[9] Lélia Blin, Shlomi Dolev, Maria Gradinariu Potop-Butucaru, and Stephane Rovedakis. Fast self-stabilizing minimum spanning tree construction - using compact nearest common ancestor labeling scheme. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010.

[10] Lélia Blin, Maria Gradinariu Potop-Butucaru, and Stephane Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *J. Parallel Distrib. Comput.*, 71(3):438–449, 2011.

[11] André Brinkmann, Sascha Effert, and Friedhelm Meyer auf der Heide. Dynamic and redundant data placement. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 29. IEEE Computer Society, 2007.

[12] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks (extended abstract). In *SPAA*, pages 119–128, 2000.

[13] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *SPAA*, pages 53–62, 2002.

[14] Augustin Chaintreau, Pierre Fraigniaud, and Emmanuelle Lebhar. Networks become navigable as nodes move and forget. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2008.

[15] Microsoft Corporation. Microsoft azure. `http://azure.microsoft.com/`, December 2014. Accessed: 2014-12-3.

[16] Toni Cortes and Jesús Labarta. Taking advantage of heterogeneity in disk arrays. *J. Parallel Distrib. Comput.*, 63(4):448–464, 2003.

[17] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[18] Peter Sheridan Dodds, Roby Muhamad, and Duncan J. Watts. An experimental study of search in global social networks. *Science*, 301:827–829, 2003.

[19] Shlomi Dolev and Ronen I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20(5):375–388, 2008.

[20] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theor. Comput. Sci.*, 410(6-7):514–532, 2009.

[21] Shlomi Dolev and Nir Tzachar. Spanders: distributed spanning expanders. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 1309–1314. ACM, 2010.

[22] George Giakkoupis and Vassos Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In Marcos Kawazoe Aguilera and James Aspnes, editors, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005*, pages 302–311. ACM, 2005.

[23] Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*, pages 596–606. IEEE, 2005.

[24] Bernhard Haeupler, Gopal Pandurangan, David Peleg, Rajmohan Rajaraman, and Zhifeng Sun. Discovery through gossip. In Guy E. Blelloch and Maurice Herlihy, editors, *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, pages 140–149. ACM, 2012.

[25] Mor Harchol-Balter, Frank Thomson Leighton, and Daniel Lewin. Resource discovery in distributed networks. In Brian A. Coan and Jennifer L. Welch, editors, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999*, pages 229–237. ACM, 1999.

[26] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In Steven D. Gribble, editor, *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*. USENIX, 2003.

[27] Thomas Hérault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, and Joffroy Beauquier. A model for large scale self-stabilization. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–10. IEEE, 2007.

[28] Google Inc. Google cloud. `https://cloud.google.com/`, December 2014. Accessed: 2014-12-3.

[29] Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In Srikanta Tirthapura and Lorenzo Alvisi, editors, *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 131–140. ACM, 2009.

[30] Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. A self-stabilizing and local delaunay graph construction. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 771–780. Springer, 2009.

[31] Junjie Jiang, Ruoyu Pan, Changyong Liang, and Weinong Wang. Bichord: An improved approach for lookup routing in chord. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam, editors, *Advances in Databases and Information Systems, 9th East European Conference, ADBIS 2005, Tallinn, Estonia, September 12-15, 2005, Proceedings*, volume 3631 of *Lecture Notes in Computer Science*, pages 338–348. Springer, 2005.

[32] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22,2003, Revised Papers*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer, 2003.

[33] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 654–663. ACM, 1997.

[34] Jon M. Kleinberg. The small-world phenomenon: an algorithm perspective. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 163–170. ACM, 2000.

[35] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: a self-stabilizing chord overlay network. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 235–244. ACM, 2011.

[36] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. A self-stabilization process for small-world networks. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1261–1271. IEEE Computer Society, 2012.

[37] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. CONE-DHT: A distributed self-stabilizing algorithm for a heterogeneous storage system. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 537–549. Springer, 2013.

[38] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. A deterministic worst-case message complexity optimal solution for resource discovery. In Thomas Moscibroda and Adele A. Rescigno, editors, *Structural Information and Communication Complexity - 20th International Colloquium, SIROCCO 2013, Ischia, Italy, July 1-3, 2013, Revised Selected Papers*, volume 8179 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2013.

[39] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: A self-stabilizing chord overlay network. *Theory Comput. Syst.*, 55(3):591–612, 2014.

[40] Sebastian Kniesburges and Christian Scheideler. Hashed patricia trie: Efficient longest prefix matching in peer-to-peer systems. In Naoki Katoh and Amit Kumar, editors, *WALCOM: Algorithms and Computation - 5th International Workshop, WALCOM 2011, New Delhi, India, February 18-20, 2011. Proceedings*, volume 6552 of *Lecture Notes in Computer Science*, pages 170–181. Springer, 2011.

[41] Sebastian Kniesburges and Christian Scheideler. Brief announcement: Hashed predecessor patricia trie - A data structure for efficient predecessor queries in peer-to-peer systems. In *Distributed*

Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings, pages 435–436, 2012.

[42] Manfred Kochen. *The Small World: A Volume of Recent Research Advances Commemorating Ithiel de Sola Pool, Stanley Milgram, Theodore Newcomb*. 1989.

[43] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In Miguel Castro and Robbert van Renesse, editors, *Peer-to-Peer Systems IV, 4th International Workshop, IPTPS 2005, Ithaca, NY, USA, February 24-25, 2005, Revised Selected Papers*, volume 3640 of *Lecture Notes in Computer Science*, pages 13–23. Springer, 2005.

[44] Sailesh Kumar, Jonathan S. Turner, Patrick Crowley, and Michael Mitzenmacher. HEXA: compact data structures for faster packet processing. In *Proceedings of the IEEE International Conference on Network Protocols, ICNP 2007, October 16-19, 2007, Beijing, China*, pages 246–255. IEEE, 2007.

[45] Shay Kutten, David Peleg, and Uzi Vishkin. Deterministic resource discovery in distributed networks. *Theory Comput. Syst.*, 36(5):479–495, 2003.

[46] Ben Leong, Barbara Liskov, and Erik D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. *Computer Communications*, 29(9):1243–1259, 2006.

[47] David Liben-Nowell, Jasmine Novak, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Geographic routing in social networks. *Proceedings of the National Academy of Sciences*, 102(33):11623–11628, August 2005.

[48] Witold Litwin. Trie hashing. In Y. Edmund Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, pages 19–29. ACM Press, 1981.

[49] Witold Litwin, Nick Roussopoulos, Gérald Lévy, and Wang Hong. Trie hashing with controlled load. *IEEE Trans. Software Eng.*, 17(7):678–691, 1991.

[50] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In Aleta Ricciardi, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 183–192. ACM, 2002.

[51] Mario Mense and Christian Scheideler. SPREAD: an adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 1135–1144. SIAM, 2008.

[52] Valentin Mesaros, Bruno Carton, and Peter Van Roy. S-chord: Using symmetry to improve lookup efficiency in chord. In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '03, June 23 - 26, 2003, Las Vegas, Nevada, USA, Volume 4*, pages 1752–1760. CSREA Press, 2003.

[53] Stanley Milgram. The small-world problem. *Psychology Today*, 1(1):61–67, 1967.

[54] Alberto Miranda, Sascha Effert, Yangwook Kang, Ethan L. Miller, André Brinkmann, and Toni Cortes. Reliable and randomized data distribution strategies for large scale storage systems. In *18th International Conference on High Performance Computing, HiPC 2011, Bengaluru, India, December 18-21, 2011*, pages 1–10. IEEE Computer Society, 2011.

[55] Alberto Montresor, Márk Jelasity, and Özalp Babaoglu. Chord on demand. In Germano Caronni, Nathalie Weiler, Marcel Waldvogel, and Nahid Shahmehri, editors, *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), 31 August - 2 September 2005, Konstanz, Germany*, pages 87–94. IEEE Computer Society, 2005.

[56] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[57] Achour Mostéfaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA*, pages 109–118, 2005.

[58] Alexei Myasnikov, Vladimir Shpilrain, and Alexander Ushakov. *Group-based Cryptography (Advanced Courses in Mathematics - CRM Barcelona)*. Advanced courses in mathematics, CRM Barcelona. Birkhauser, 2008 edition, 7 2008.

[59] Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. *Theor. Comput. Sci.*, 512:119–129, 2013.

[60] Melih Onus, Andréa W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007.

[61] Amazon Web Services Inc. or its affiliates. Amazon ec2. `http://aws.amazon.com/de/ec2/`, November 2014. Accessed: 2014-11-24.

[62] Amazon Web Services Inc. or its affiliates. Amazon ec2 service level agreement. `http://aws.amazon.com/de/ec2/sla/`, November 2014. Accessed: 2014-11-24.

[63] Mihai Patrascu and Mikkel Thorup. Time-space trade-offs for predecessor search. pages 232–240, 2006.

[64] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[65] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3):241–280, 1999.

[66] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief announcement: prefix hash tree. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, page 368. ACM, 2004.

[67] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *OPODIS*, pages 88–102, 2005.

[68] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in structured P2P systems. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22,2003, Revised Papers*, volume 2735 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2003.

[69] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[70] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[71] Jose Renato Santos and Richard R. Muntz. Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In Wolfgang Effelsberg and Brian C. Smith, editors, *Proceedings of the 6th ACM International Conference on Multimedia '98, Bristol, England, September 12-16, 1998.*, pages 303–308. ACM, 1998.

[72] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hash tables. In Phillip B. Gibbons and Paul G. Spirakis, editors, *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 218–227. ACM, 2005.

[73] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology P2P systems. In Germano Caronni, Nathalie Weiler, Marcel Waldvogel, and Nahid Shahmehri, editors, *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), 31 August - 2 September 2005, Konstanz, Germany*, pages 39–46. IEEE Computer Society, 2005.

[74] Haiying Shen and Cheng-Zhong Xu. Hash-based proximity clustering for efficient load balancing in heterogeneous DHT networks. *J. Parallel Distrib. Comput.*, 68(5):686–702, 2008.

[75] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[76] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[77] Marcel Waldvogel, George Varghese, Jonathan S. Turner, and Bernhard Plattner. Scalable best matching prefix lookups. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, page 312. ACM, 1998.

[78] Jie Wang and Zhijun Yu. A new variation of chord with novel improvement on lookup locality. In Hamid R. Arabnia, editor, *Proceedings of the 2006 International Conference on Grid Computing & Applications, GCA 2006, Las Vegas, Nevada, USA, June 26-29, 2006*, pages 18–24. CSREA Press, 2006.

[79] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[80] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[81] Shu-Yuen Didi Yao, Cyrus Shahabi, and Roger Zimmermann. Broadscale: Efficient scaling of heterogeneous storage systems. *Int. J. on Digital Libraries*, 6(1):98–111, 2006.

[82] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.