Dynamics and Efficiency in Topological Self-Stabilization

PhD Thesis

Andreas Koutsopoulos
University of Paderborn

Reviewers:

1. Prof. Dr. Christian Scheideler (advisor)
2. Prof. Dr. Friedhelm Meyer auf der Heide

Στον Σάββα

Zusammenfassung

Die Probleme, die in dieser Dissertation behandelt werden, haben ihren Ursprung im Gebiet der verteilten Systeme und insbesondere der Overlay Netzwerke. Genauer gesagt untersuchen wir, wie wir die dynamische Natur der Overlay Netzwerke ”zähmen” können. Wir fragen uns: ”Wie können wir es schaffen, dass Teilnehmer ständig das Netzwerk verlassen und eintreten können, ohne dass der Zusammenhang des Netzwerkes gefährdet ist.” Eine andere Frage bezüglich der dynamischen Natur der Overlay Netzwerke, die wir uns stellen ist: ”Wie können wir garantieren, dass ein Netzwerk, was momentan eine schlechte Struktur hat, in eine erwünschte Struktur übergeht, und zwar möglichst schnell und effizient?” Um diese Frage zu beantworten, untersuchen wir die Eigenschaft der topologischen Selbststabilisierung in Overlay Netzwerken. Ein Netzwerk besitzt diese Eigenschaft, wenn es seine Struktur (oder Topologie) anpassen kann und eine erwünschte Struktur erreichen kann, unabhängig von der initialen Topologie, die das Netzwerk hat. Dies wird durch bestimmte Protokolle realisiert (selbststabilisierende Protokolle), die ständig an jeden Teilnehmer des Netzwerkes lokal laufen, was heißt, dass jeder Knoten Wissen nur über seine direkte Nachbarn im Netzwerk verfügt und Aktionen gemäß dieses Wissens durchführt.

In dieser Dissertation stellen wir effiziente selbststabilisierende Protokolle für diverse Topologien vor. Diese Topologien haben gewisse Eigenschaften, die für diverse Anwendungen erwünscht sind. Die Qualität der Effizienz unser Protokolle messen wir anhand der Anzahl der Bits, die durch das Protokoll gesendet werden, als auch die Zeit, die benötigt wird, um die erwünschte Topologie zu erreichen.

Wir haben erwähnt, dass wir auch das Verwalten der Situation untersuchen wollen, in der Teilnehmer (oder Knoten) das Netzwerk verlassen wollen. In dieser Dissertation untersuchen wir dieses Problem und geben eine formelle Definition davon. Wir untersuchen unter welchen Umständen das Problem lösbar ist und stellen letztendlich ein algorithmisches Framework vor, welches an existierenden verteilten Protokollen angewendet werden kann, um dieses Problem zu lösen.

Abstract

The problems studied in this thesis originate from the field of distributed systems and in particular overlay networks. More specifically, we study how to handle the dynamic nature of overlay networks. We ask ourselves the question: ”How can we handle the situation in which participants can constantly join and leave the network, without endangering that the network disconnects?” Another question we ask ourselves regarding the dynamic nature of overlay networks is: ”How can we guarantee that a network which currently has a structure that is in a bad state, recovers to a desired structure quickly and efficiently?” In order to answer that question, we study topological self-stabilization in overlay networks. This is a property that a network has, if it is able to adapt its structure (or topology) and reach a desired topology, independently of the original topology that the network has. This is done by specific protocols (self-stabilizing protocols), which are run constantly on each network participant and are local: i.e., each node has knowledge only about its direct neighbors in the network and takes actions according to that knowledge.

In this thesis we present efficient self-stabilizing protocols for several topologies. These topologies have specific properties that are desired for various applications. The quality of the efficiency of our protocols is measured according to communication bits that are sent, due to our protocols, as well as according to the time needed in order for the desired topology to be reached.

We mentioned above that we want to examine the case of handling participants (or nodes) that are leaving the network. In this thesis we study this problem and give a formal definition of it. We investigate under which conditions this problem is solvable and in the end give a framework which can be applied by existing distributed protocols, with certain prerequisites, in order to solve this problem.

Acknowledgements

I would like to thank first of all my advisor Christian Scheideler, who introduced me very interesting problems and gave me the correct insights for achieving solutions for many of them. I also want to give many thanks to my colleagues Sebastian Kniesburges and Thim Strothmann, who are the co-authors of the papers presented in this thesis. The countless hours we spent at meetings just thinking, presenting ideas and exchanging arguments was an invaluable experience. Of course I would like also to thank the rest of my colleagues of the Theory of Distributed Systems group, who helped me every time I had asked for advice, ideas and any other kind of help. The same holds for the colleagues of the other theory groups, whose participation in the weekly Theory Seminar made the time at work even more interesting. Lastly, I would like to thank my family and friends, and especially my wife Caro for having patience with me and supporting me through all these years.

Contents

Prologue

Conducting data storage and computing in a distributed manner has become a common practice, due to the nature of many applications. The number and variety of client devices is increasing over the years, which results in an increasingly complex array of end points to serve. The use of social, mobile, and embedded technology causes the amount and variety of data collected to expand exponentially. So the need to mine this data for insights -for businesses or other organizations- is becoming imperative.

Distributed systems have applications that vary from Distributed File Systems (Hadoop [85]), peer-to-peer based protocols (BitTorrent [86]), cloud networks, to volunteer-based projects (Folding@home, SETI@ home ([64] [74]), to name a few examples.

The peer-to-peer distributed computing model is described by a distributed system, where participants are allowed to leave or enter the system at any time and are aware only of their local view of the system. This model ensures uninterrupted uptime and access to applications and data even in the event of partial system failure. Vendors guarantee very high availability, a feature which cannot be easily be matched using centralized computing.

Distributed systems are highly dynamic by nature. In peer-to-peer systems the participants can join and leave any time and also connections might fail. In cloud-based systems, where different cloud providers may be used for one application, the unavailability of some providers is also an issue, due to the dynamic nature of computation or storage needs (scaling up and down over time). This establishes a high degree of dynamics. The higher this degree is over a period of time, the more probable it is for undesired situations to occur in the distributed system. The structure of the (overlay) network of the system might lose its original topology and no longer maintain the desired qualities such as good expansion, or low diameter, for example. In the worst case, parts of the network can disconnect from each other.

We therefore need protocols which can adapt to these dynamics and help the system to recover from such undesired situations. The participants in a distributed system conduct local computations: i.e., they do not have a global view on the whole system. Thus, protocols are needed that run on the participants of the system (by considering only the local view of the participant) and can achieve that the whole system self-recovers from such undesired situations automatically. For that reason we use *self-stabilizing* protocols. Self-stabilization is the property of a system to recover from any undesired state and maintain that state. A more formal definition of self-stabilization will be given in the next section.

Note that simple self-repairing mechanisms, which are used in many distributed protocols, do not suffice to ensure recovery out of any undesired state (or network structure in case of overlay networks). For example, it was shown ([87]) that several invariants presented for the original Chord protocol ([80]) (one of the most influential papers in the field of overlay networks) are not correct: i.e., they do not suffice in order for the network to recover out of any faulty state.

In this thesis, several self-stabilizing protocols which maintain certain network structures are presented. These structures (or overlay topologies) are useful for implementing DHTs (distributed hash tables) or for simply providing overlays with some desired properties. We will focus on giving efficient protocols, concerning the time and communication overhead of these protocols.

We will furthermore focus on one aspect of the dynamics of overlay networks, the exiting of participants from the system, since this aspect endangers the connectivity if not handled carefully. We will examine what exactly describes a departure, when is a participant able to leave the system without endangering connectivity, and how we can solve this problem. Moreover, we will study the case of constructing an algorithmic framework that can be applied to distributed protocols, in order for them to be able to handle departures of participants.

1 Preliminaries

1.1 The Model

We assume that a distributed system consists of a fixed set of participants (or nodes) with fixed identifiers, ids for short, that are globally ordered. We refer to nodes and their identifiers interchangeably. The system is controlled by an algorithm (or protocol) that specifies the variables, and actions that are available in each node. In addition to the algorithm-based variables there is a system-based variable called a *channel* whose values are sets of messages. The channel message capacity is unbounded, and messages will never get lost. We consider point-to-point communications (multi-cast and broadcast primitives are not considered). We treat all messages sent to a node u as belonging to a single incoming channel $u.C$.

The nodes carry out computations and interact with each other by performing *actions*. There are two types of actions. The first type of action has the form of a standard procedure $\langle label \rangle$ $(\langle parameters \rangle)$: $\langle command \rangle$, where $label$ is the unique name of that action, $parameters$ specifies the parameter list of the action, and $command$ specifies the statements to be executed when calling that action. Such actions can be called remotely. In fact, we assume that every message must be of the form $\langle label \rangle(\langle parameters \rangle)$, where $label$ specifies the action to be called in the receiving node and $parameters$ contains the parameters to be passed to that action call. All other messages will be ignored by the nodes. Apart from being triggered by messages, these actions may also be called locally by the node, that causes their immediate execution. The second type of action has the form $\langle label \rangle$: $\langle guard \rangle \longrightarrow \langle command \rangle$, where $label$ and $command$ are defined as above and $guard$ is a predicate over local variables. We call an action whose guard is simply **true** a *timeout* action, which is an action that is executed periodically, with the period being a specific time interval.

Each node stores a number of variables, and the combination of the values of its variables determines its *state*. The *system state* is the combination of the states of all nodes as well as the content of all channels. An action in some node u is *enabled* in some system state if its guard evaluates to **true** or its guard detects the presence of a particular message type in $u.C$. The action is *disabled* otherwise.

A *computation* is an infinite fair sequence of states such that for each state s_i, the next state s_{i+1} is obtained by executing an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. We assume two kinds of fairness of computation: weak fairness of action execution and fair message receipt. *Weak fairness* of action execution means that if an action is enabled in all but finitely many states of the computation, then this action is executed infinitely often.

Fair message receipt means that if the computation contains a state where there is a message in a channel of a node that is not gone, this computation also contains a later state where this message is not present in the channel: i.e., the message is received.

A *computation suffix* is a sequence of computation states past a particular state of this computation. In other words, the suffix of the computation is obtained by removing the initial state and finitely many subsequent states. Note that a computation suffix is also a computation.

We consider algorithms that do not manipulate the internals of node identifiers. Specifically, an algorithm is a *copy-store-send* algorithm if the only operations that it executes on node *ids* is copying them, storing them in local node memory and sending them in a message. That is, operations on *ids* such as addition, radix computation, hashing, etc. are not used. In a copy-store-send algorithm, if a node does not store an *id* in its local memory, the node may learn this *id* only by receiving it in a message. A copy-store-send algorithm cannot introduce new *ids* to the system. It can only operate on the *ids* that are already there. We will use the term algorithm and protocol interchangeably.

We model the network by a directed graph $G = (V, E)$. The set V represents the nodes of the network. The set of edges E describes the possible communication pairs. E consists of two subsets: the *explicit edges* $E_e = \{(u, v) : v \text{ is stored as a variable of } u\}$ and the *implicit edges* $E_i = \{(u, v) : v \text{ is part of a message in } u\text{'s channel } (v \in u.C)\}$, So it holds that $E = E_e \cup E_i$. Moreover, we define $G_e = (V, E_e)$.

There are different models of synchronicity for message passing systems. In *asynchronous* systems there is no guarantee of when a message sent by a node u will arrive at its destination v. There exist also *partially synchronous* message passing system models, where there is usually an upper time bound on when a message is certain to be delivered, but this time bound is not known to the participants of the network. In the *synchronous* case however, the participants have this knowledge. So in this case the protocols operate in synchronous rounds. We assume that in the time period of one round a node u can process the messages in $u.C$ and conduct the desired actions. We furthermore assume that messages sent in one round by the nodes in the network are delivered before the beginning of the next round. In this thesis we will also refer to rounds as time steps. For the first part of the thesis we consider the asynchronous message passing model, whereas for the second part the synchronous message passing model is considered.

1.2 Notation

We denote the id of a node u as $u.id$. We assume that the ids are values within the interval $(0, 1)$. More precisely, the id of a node u, $u.id$ is chosen uniformly at random from that interval. Besides that, a node maintains several variables stored in its local memory. These are called node variables. A node variable a maintained by a node u is denoted as $a.u$. Global variables are variables that are not stored or used by the nodes. Usually, these are state properties of the nodes or the network and are used for proof purposes. A global variable b that applies to a property of node u is denoted as $b(u)$. The values of global variables are fixed, whereas the values of node variables can be changed through the actions of the protocol. The set of all node variables of a node u (i.e., the set of outgoing explicit edges) is also called the *neighborhood* of u and is denoted as $u.N$.

A message in general consists of the following parts: a *sender id*, which is the id of the node sending the message, one or more optional *additional ids*, if the sender wants to inform the receiving node about another node, and the *type* of the message. When we want to denote that a node u sends a message m to node v, we write: u : *send* $message(m.id, m.id_2, ..., m.id_k, m.type)$ to v, where $m.id, m.id_2, ..., mid_k$ are the *additional ids* sent through the message and $m.type$ is the message type. We refer to the sender of the message as $m.sender$. Note that there can be arguments of other types in a message, other than ids. The message type is always given as the last argument. Note that if it holds for the receiver of a message v that $v = null$, we then consider that no message is sent.

We say that an event happens with high probability (w.h.p.), if it happens with probability $p(n)$, such that $p(n) \to 1$ for $n \to \infty$.

We say that a node u is *greater* or *at the right* (resp. *smaller* or *at the left*) of another node v if $u.id > v.id$ (resp. $u.id < v.id$). Note that we also may write $u < v$ (resp. $u > v$) instead of $u.id < v.id$ (resp. $u.id > v.id$). The *distance* between two nodes u and v, ($distance(u, v)$) with $u.id > v.id$ is defined as $min\{|u.id - v.id|, |1 - u.id + v.id|\}$. We say that a node u is *closer* to a node v than to a node w if $distance(u, v) < distance(v, w)$. We say that a node u is *between* node v and node w if $v.id < u.id < w.id$ (or $w.id < u.id < v.id$).

We say that two nodes u and v in a network graph $G = (V, E)$ are *contiguous* if $u = \operatorname{argmin}_{w \in V} \{w.id > v.id\}$ or if $u = \operatorname{argmin}_{w \in V} \{w.id\} \wedge v = \operatorname{argmax}_{w \in V} \{w.id\}$.

The *length* of a path of edges p, i.e. $length(p)$, is equal to the number of nodes it contains minus one. Let $u_{min} = \operatorname{argmin}_{v \in p} \{v.id\}$ and $u_{max} = \operatorname{argmax}_{v \in p} \{v.id\}$ then the *range of a path* ($range(p)$) is given by $range(p) = u_{max} - u_{min}$.

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

1.3 Oracles

An *oracle* $\mathcal{O}$ is a predicate that depends on the system state and the node calling it. We restrict our attention to oracles that *only* depend on the current network graph: i.e., oracles are of the form $\mathcal{O}$: $\mathcal{GS} \times V \to \{$**true**, **false**$\}$ where $\mathcal{GS}$ is the set of network graph topologies and V is the set of nodes. We assume that the oracles we introduce always work correctly when used: i.e. oracle $\mathcal{O}$ answers **true** iff the property the oracle is queried for indeed holds.

2 Self-Stabilization

The term self-stabilization refers to the property of a system to be able to recover out of any arbitrary state and eventually converge to a desired state. An algorithm is said to be self-stabilizing if it can deliver that requirement. More precisely:

A protocol is *self-stabilizing* if it satisfies the following two properties.

Convergence: starting from an arbitrary system state, the protocol is guaranteed to arrive at a legitimate state.

Closure: starting from a legitimate state, the protocol remains in legitimate states thereafter.

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

2.1 Topological Self-Stabilization

Topological self-stabilization refers to overlay networks and the goal is to state a protocol that reaches a desired network topology starting from an (almost) arbitrary initial network topology.

More formally, we want to construct a protocol *P* that *solves* an overlay problem *OP* starting from an initial topology of the set *IT*. 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 in this state covers the whole network, and there are no identifiers that do not 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 also call the program states in *OP legal states* or *stable 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 *P* started with any state in *IT* reaches a legal state in *OP*. *Closure* means that *P* started in a legal state in *OP* maintains a legal state (see Figure 2.1). In this thesis only protocols will be considered, where the stable state is unique (i.e. the number of target topologies is 1).

For each node variable $u.a$ of a node u we define $a(u)$ to be the global variable that has the value $u.a$ would have at the stable state. Note that a network is in a stable state if for every node variable $u.a$ of a node u it holds that $u.a = a(u)$.

We say that a node variable $a.u$ is *valid* if it does not locally violate the restrictions set by the protocol. For example, if the variable $u.successor$ is defined through the protocol as a node w such that $w.id > u.id$, but it currently is the case that $(u.successor).id < u.id$, then the variable $u.successor$ is invalid. We also say that a message contains invalid information, if the content in this message does not depict the truth. For example it contains the information that $v.(u.state) = active$, (for some variable of state of a node u), whereas it holds that $state(u) = active$.

Definition 2.1
We say that an overlay network is in a ***valid*** *state iff all node variables are valid and there are no messages containing invaliding information in the system.*

2.2 Complexity Measures

In order to study the performance of our algorithms we adopt the complexity measures used in the literature. The *self-stabilization time* of a protocol in the synchronous message passing system is the number of rounds it takes in the worst case for the protocol to reach a stable state. The *message complexity* of the self-stabilization procedure is the number of messages sent in the network during the *self-stabilization time*. The *message complexity* of the stable state refers to the number of messages sent in the network per round if the network is in the stable state. The *communication complexity* refers to the number of communication bits sent in the network. Note that in our case, the messages sent through our protocols contain just a constant number of node ids plus some other arguments of constant size. Since each node id requires $\mathcal{O}(\log n)$ space, each message contains $\mathcal{O}(\log n)$ communication bits. We will therefore restrict ourselves to the study of message complexity, since the communication complexity would just be multiplying the message complexity by $\log n$. In the rare cases where the messages sent by the protocols contain more than a constant number of ids we will count just these messages multiple times in order to compute the message complexity.

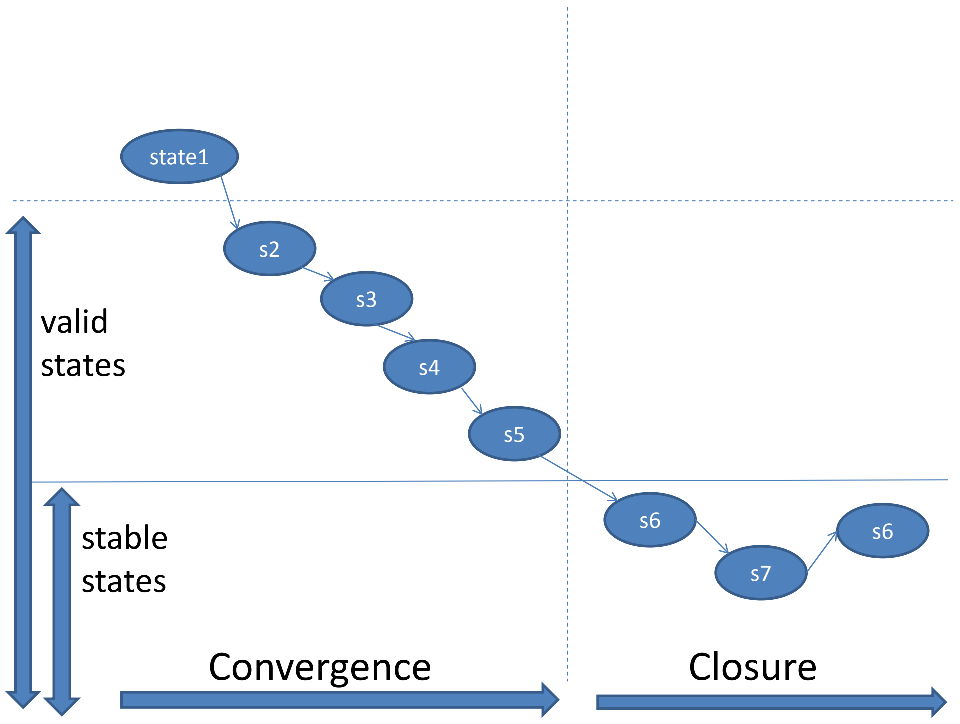
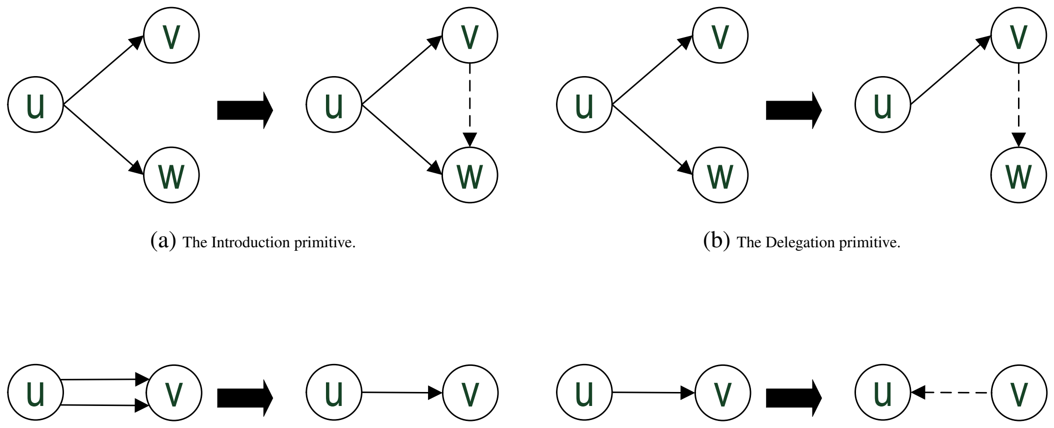


Figure 1: Illustration of the convergence and the closure phase. The protocol *P* eventually reaches the set of stable states *OP*, which in this case are states $s6$ and $s7$.

2.3 Topological Self-stabilization in the Literature

There is a large body of literature on how to efficiently maintain overlay networks, e.g., [51, 52, 53, 25, 54, 55, 56, 57, 24, 2, 12]. While many results are already known on how to keep an overlay network in a legal state, far less is known about self-stabilizing overlay networks. The idea of self-stabilization in distributed computing firstly appeared in a classical paper by E.W. Dijkstra in 1974 [1] in which he looked at the problem of self-stabilization in a token ring. Interestingly, although 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, [26, 37]. The Iterative Successor Pointer Rewiring Protocol [26] and the Ring Network [37], for example, organize the nodes in a sorted ring. In [27] Dolev and Kat describe a strategy to build a hypertree with a polylogarithmic degree and search time. In [35], Onus et al. present a local-control strategy called linearization for converting an arbitrary connected graph into a sorted list. Various self-stabilizing algorithms for different network overlay structures have been considered over the years [32, 30, 28, 29, 27]. Jacob et al. [32] generalize insights gained from graph linearization to two dimensions and present a self-stabilizing construction for Delaunay graphs. In another paper, Jacob et al. [30] 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 [28] and [29] Dolev and Tzachar show self-stabilizing algorithms for forming sub-graphs such as clusters or expanders in just a polylogarithmic number of rounds.



(a) The Introduction primitive.

(b) The Delegation primitive.

(c) The Fusion primitive.

(d) The Reversal primitive.

Figure 2: The four primitives. The dashed edges represent implicit edges, whereas the normal edges represent explicit edges.

3 Basic Overlay Network Operations

Overlay network protocols in general work by manipulating the edges of the network graph in specific ways. First we will introduce some primitives for edge manipulation in overlay networks. These primitives (in combination) most importantly ensure the maintenance of weak connectivity, a property that is very important for overlay management protocols, and for topological self-stabilization protocols in particular.

3.1 The primitives for Edge Manipulation

Here we introduce four primitives for manipulating edges in an overlay network that are safe in a sense that they preserve weak connectivity (as long as there is no fault). This implies that *any* distributed protocol whose actions can be decomposed into these four primitives is guaranteed to preserve weak connectivity. The four primitives are:

Introduction If a node u has a reference to two nodes v and w, u *introduces* w to v by sending a message to v containing a reference to w while keeping the reference to w.

Delegation If a node u has a reference to two nodes v and w, then u *delegates* w's reference to v by sending a message to v containing a reference to w and deletes the reference to w.

Fusion If a node u has two references v and w with $v = w$, then it *fuses* them by keeping only one of these references.

Reversal If a node u has a reference to some other node v, then it *reverses* the connection by sending a reference of itself to v and deleting the reference to v.

A special case of introduction is the *self-introduction*, where u sends a reference of itself to v, but does not delete its reference to v. The four primitives have the advantage that they can be executed locally by every node in a wait-free fashion (as none of the primitives requires an acknowledgement). Also, they just need the ability to check whether two references point to the same node (see the Fusion) to be implementable. Other than that, access to the contents of the references is not needed, which is useful, for example, for anonymous networks. Moreover, it holds:

Lemma 3.1 *Introduction, Delegation, Fusion, and Reversal preserve weak connectivity.*

Proof. The statement obviously holds for Introduction since only additional edges are introduced. In Delegation an edge (u, w) is deleted, but there still exists a path from u to w via v, so u and w are still in the same weakly connected component. Fusion deletes an edge only if it is superfluous for weak connectivity. The Reversal rule deletes an edge (u, v) but replaces it with an edge (v, u), thereby also preserving weak connectivity. □

Let $\mathcal{P}$ denote the set of all distributed protocols where all interactions between nodes can be decomposed into the four primitives. Not surprisingly, all of the self-stabilizing topology maintenance protocols proposed so far (e.g., [65, 66, 69, 73, 30, 2]) satisfy this property (as otherwise they would risk disconnection). Lemma 3.1 implies that any protocol in $\mathcal{P}$ preserves weak connectivity, which was previously shown individually for each cited protocol. Note that the first three primitives even preserve strong connectivity in a sense that for any pair of nodes u, v with a directed path in the network graph G there will always be a directed path from u to v in G when only allowing these three primitives. We say that a set of primitives is *universal* if the primitives allow one to get from any weakly connected graph $G = (V, E)$ to any other weakly connected graph $G' = (V, E')$ The set is *weakly universal* if G' is strongly connected.

Theorem 3.2 *Introduction, Delegation, Fusion, and Reversal are universal.*

Proof. We give a general strategy how to transform an arbitrary weakly connected graph $G = (V, E)$ into any other weakly connected graph $G' = (V, E')$. At first, note that by continuously introducing all neighbors of every node to each other, including self-introduction, then the topology of G is eventually transformed from G into a clique (in fact, $\mathcal{O}(\log n)$ rounds of communication are sufficient for that as the distances between the nodes are essentially cut in half in each round).

Next we show that by using Delegation and Fusion, one can transform G from the clique to the bi-directed extension $G'' = (V, E'')$ of G', i.e., the graph where for any edge $(u, v) \in E'$ there are edges $(u, v), (v, u) \in E''$. To do so, we make use of the fact that G'' is strongly connected. Consider an arbitrary edge (u, w) in G that is not in E''. Since G'' is strongly connected, there exists a shortest path from u to w in G'' and therefore also in G (as we first want to keep all edges in G''). Let v_1 be the first neighbor of u along that shortest path. Then u delegates the reference of w to v_1. Now the node v_1 (and all other nodes on the shortest path) proceed similar to u by forwarding the reference to w along the shortest path up to the last node v_k, who is a neighbor of w. Node v_k uses Fusion to merge the edge with $(v_k, w) \in E''$. By applying this procedure to all edges not in E'', all that remains is G''.

At last we can use Reversal and Fusion to get from G'' to G'. To do so, every edge (u, v) that is in E'', but not in E' is reversed by u. Then the newly created edge (v, u) is fused with the already existing edge $(v, u) \in E'$. □

Note that Theorem 3.2 only shows that *in principle* it is possible to get from any weakly connected graph to any other weakly connected graph. From the proof we can conclude the following corollary.

Corollary 3.3 *Introduction, Delegation, and Fusion are weakly universal.*

Furthermore, Introduction, Delegation, Fusion and Reversal are not only sufficient for universality but also necessary, i.e., by removing one primitive, universality is lost.

Theorem 3.4 *Introduction, Delegation, Fusion and Reversal are necessary for universality.*

Proof. To prove the lemma, we show that each primitive has a unique function that cannot be replaced by the other primitives. Introduction is the only primitive that can create new edges, so without it, any Graph G' with $|E'| > |E|$ cannot be reached from G. Fusion is the only primitive that reduces the overall number of edges. Delegation is necessary, since by using only Introduction, Fusion and Reversal, a protocol can never locally disconnect two specific nodes. Finally, consider an example graph G consisting of two nodes u and v and an edge (u, v). Reversal is necessary to reach the goal topology G' that consists solely of the edge (v, u). $\square$

4 Thesis Outline

The thesis is divided into two parts. The first part focuses on the problem of node departures in distributed systems. We give self-stabilizing solutions for special node departure problems and end by giving a general self-stabilizing algorithmic framework for solving the *Finite Departure Problem*, which we call the *FDP*. The protocols are developed for the asynchronous message passing system model. The content of the first part is based on the following papers: ([71, 78])

Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, Thim Strothmann: On Stabilizing Departures in Overlay Networks. SSS 2014.

Andreas Koutsopoulos, Thim Strothmann and Christian Scheideler: Towards a Universal Approach for the Finite Departure Problem in Overlay Networks, SSS 2015.

In the second part we study specific self-stabilizing problems for certain goal topologies. We focus on giving *efficient* solutions to these problems, in terms of message complexity and self-stabilization time. The setting considered in this part is synchronous message passing. The content of the second part is based on the papers we wrote on topological self-stabilization. ([33, 77, 76, 75])

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

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

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler: CONE-DHT: A Distributed Self-Stabilizing Algorithm for a Heterogeneous Storage System. DISC 2013

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler: A Deterministic Worst-Case Message Complexity Optimal Solution for Resource Discovery. SIROCCO 2013,Theoretical Computer Science 584

Part I

On Stabilizing Departures in Overlay Networks

A fundamental problem for peer-to-peer systems is to maintain connectivity while nodes are leaving, i.e., the nodes requesting to leave the peer-to-peer system are excluded from the overlay network without affecting its connectivity. There are a number of studies for safe node exclusion if the overlay is initially in a well-defined state initially. Surprisingly, the problem has not been formally studied yet for the case in which the overlay network is in an arbitrary initial state: i.e., we are looking for a *self-stabilizing* solution for excluding leaving nodes.

We begin in the first chapter by formally defining the node departure problem in distributed systems as the *Finite Departure Problem ($\mathcal{FDP}$)* and investigate when this problem can be solved. Unfortunately, in the general case there is no self-stabilizing distributed algorithm for the $\mathcal{FDP}$ that works without using oracles. We give an algorithm that provides a solution with the help of an oracle called $\mathcal{NIDEC}$. Note that in this part of the thesis we consider asynchronous message passing systems.

We then relax the $\mathcal{FDP}$ to the $\mathcal{FSP}$ (*Finite Sleep Problem*), in order for it to be solvable without the use of oracles. In the $\mathcal{FSP}$, the leaving decision does not have to be final: the nodes may resume computation if necessary.

The results of the first chapter are based on the paper

Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, Thim Strothmann: On Stabilizing Departures in Overlay Networks. SSS 2014.

In the second chapter we further investigate the $\mathcal{FDP}$ problem. We provide a self-stabilizing algorithm that solves the problem for the case of anonymous networks and also give a general algorithmic framework which can be applied to a large class of distributed algorithms in order to solve $\mathcal{FDP}$. The content of the chapter is based on the paper

Thim Strothmann, Andreas Koutsopoulos and Christian Scheideler
Towards a Universal Approach for the Finite Departure Problem in Overlay Networks, SSS 2015.

Chapter 1

Investigating Departures

1 Introduction

Peer-to-peer systems allow computers to interact and share resources without the need for a central server or centralized authority. This ability to self-organize has made peer-to-peer systems very popular. Since participation in such systems is usually voluntary, the peers may arrive and depart at any time. A peer may even leave the network without notice. Therefore, maintaining a connected overlay network is a challenging task. Many strategies help to alleviate this problem. They include using an overlay network with a high expansion or separating the peers into more reliable super-peers forming an overlay network on behalf of the other peers that just connect to one or more super-peers. While these strategies may work well in practice, rigorous research on when it is safe to leave the network is still in its infancy. In this chapter we lay the foundation for a rigorous treatment of node departures in the context of self-stabilization. In fact, we are the first to provide answers to the following question:

Is it possible to design a distributed algorithm that allows any collection of nodes to eventually leave a network from any initial state without losing connectivity?

Self-stabilization makes the above question non-trivial. A self-stabilizing algorithm recovers from an arbitrary initial state. Hence, a self-stabilizing node departure algorithm has to handle the states where the departing node is about to leave and may disconnect the network.

1.1 Related Work

The difficulty of the Finite Departure Problem resembles that of fault-tolerant agreement in distributed systems. Fault-tolerant agreement has been studied in the context of the famous Consensus Problem. It is shown [70] that the problem is not solvable in an asynchronous system even if only a single node may crash. However, solutions to the *stabilizing consensus problem* are known [61, 68], in which it is not required that each node irrevocably commits to a final value but that eventually they arrive at a common, stable value without being aware of that. The impossibility can also be circumvented by the use of specialized oracles known as failure detectors [67].

Due to the popularity of peer-to-peer networks, the research literature on this subject is extensive [60, 51, 52, 53, 25, 54, 56, 24, 2]. While departure algorithms have been proposed in these papers, none of them are self-stabilizing. In fact, a rigorous treatment of when it is safe to leave the system has not been yet attempted. Cases in which the rate of churn is limited have already been considered [59, 72, 79]. Kuhn et al [59, 72, 79] handle this limitation by organizing the nodes into cliques of $\Theta(\log n)$ size that they call super-nodes. Hayes et al. [72] handle limited churn with a topological repair strategy called Forgiving Graph. For the case that the nodes have a sufficient amount of time to react, Saia et al. [83] propose a network maintenance algorithm called DASH to repair the network resulting from an arbitrary number of deletions. Limited churn has also been studied in the context of adversarial nodes [62, 63, 84]. While there is no work on self-stabilizing node departures, several self-stabilizing peer-to-peer algorithms are proposed. However, none of these provide any means to exclude nodes that want to leave the network.

2 Formalizing the Problem

Before we define a legitimate state for the problems considered in this chapter, we restrict the set of initial states to exclude trivially useless parts of that state. But first we introduce some notation, as well as our specific model.

2.1 Model and Notation

We call nodes that want to depart the network *leaving* nodes. The rest are *staying* nodes. Each node has a read-only boolean variable called *leaving*. If this variable is **true**, the node is *leaving*; the node is *staying* otherwise.

We place no bounds on message propagation delay or relative process execution speeds: i.e., we allow fully asynchronous computations and non-FIFO message delivery.

There are two special commands that are important for the study of our finite departure problem, **exit** and **sleep**. If a node executes **exit** it enters a designated *exit state*. We call such a node *gone*. If a node executes **sleep**, it enters a *sleep state*. Such a node is *asleep*. If a node never wakes up again, it is called *permanently asleep*. A node that is neither gone nor asleep is called *awake*.

In this part of the thesis, an action in some node p is considered *enabled* in some system state if its guard evaluates to **true** and p is awake, or there is a message in $p.C$ requesting to call it and p is awake or asleep. In the latter case, p becomes awake again as soon as the corresponding message is processed (in which case it is removed from $p.C$). The action is *disabled* otherwise. Hence, while a gone node never wakes up again, an asleep node may wake up again when processing an appropriate message.

A *(weakly) connected component* in some directed graph G is a sub-graph of G of maximum size so that for any two nodes u and v in that sub-graph there is a (not necessarily directed) path from u to v. Two nodes that are not in the same weakly connected component are *disconnected*. We call a node p *hibernating* if p is asleep, $p.C$ is empty, and all nodes q that have a directed path to p in the network graph (which we here will call PG) are also asleep and $q.C$ is empty.

Proposition 2.1 *For any copy-store-send algorithm and any system state of that algorithm in which node p is hibernating, p is permanently asleep.*

Proof. Let $PG(p)$ be the sub-graph containing all nodes q with a directed path to p. A node q in $PG(p)$ can only be woken up by a message, but such a message would have to come from a node q' outside of $PG(p)$, which would require an edge (q', q) in PG. Since such an edge does not exist, the proposition follows. □

Initially gone nodes are useless as they will never perform any computation. Hence, we assume that the initial state only consists of non-gone and non-hibernating nodes. We also restrict the initial state to contain only messages that can trigger an action since the other messages will be ignored.

2.2 Problem Statement

Since we consider node departures, we define a system state to be *legitimate* if (i) every staying node is awake, (ii) every leaving node is gone and (iii) for each weakly connected component of the initial graph, the staying nodes in that component still form a weakly connected component. To do this the nodes are allowed to use the **exit** command described above (the **sleep** command will be considered later). Now we are ready to formally state our problem.

***Finite Departure Problem* ($\mathcal{FDP}$)** : Eventually a legitimate state is reached for the case that the **exit** command is available to the nodes (and not the **sleep** command) .

A self-stabilizing solution for this problem must be able to solve it from any initial state and to satisfy the closure property afterwards. Notice that (i) and (ii) can trivially be maintained in a legitimate state, so for the closure property one just needs to ensure that (iii) is also maintained.

In the following, a node is called *relevant* if it is neither gone nor hibernating. Otherwise we call it *irrelevant*. Since hibernating and gone nodes will never execute any action, for the self-stabilization we only consider initial states in which all nodes are relevant. Also we do not consider initial states, where inactive nodes or ids not corresponding to existing nodes are present. Their handling would require failure/presence detectors which is beyond the scope of this thesis. From now on, an initial system state satisfies all of these constraints.

A node p can *safely* leave a system if the removal of p and its incident edges from PG does not disconnect any relevant nodes.

3 Investigating the Problem

The question that arises is whether $\mathcal{FDP}$ is solvable by a distributed protocol. As we will show, in the general case not. In fact, in order to make $\mathcal{FDP}$ to be solvable, we would have to make many restrictions to the network graph. It does not suffice to have a strongly connected graph, even in the synchronous case. It is possible only if we make immensely strong restrictions, for example that there are no incoming edges to a leaving node u, of which u is to aware of, or that the graph without the leaving nodes is still connected. Otherwise one could always construct instances, such that the problem is not solvable.

Theorem 3.1 *There is no distributed self-stabilizing solution to the $\mathcal{FDP}$.*

Proof. Assume that algorithm $\mathcal{A}$ is a self-stabilizing solution to the $\mathcal{FDP}$. We consider the following counterexample. Consider a system of at least three nodes. The computation of $\mathcal{A}$ starts in a state where all nodes but one, node v, are weakly connected. Hence, v remains disconnected from the system for the rest of the computation. Among the connected nodes, u is leaving. Since $\mathcal{A}$ is a solution to the $\mathcal{FDP}$, the computation will eventually reach a state s_1 in which u calls **exit** in some action A enabled in s_1.

We take s_1 and construct another state s_2 where there is a message carrying the id of v in the incoming channel of node u. In s_2, all nodes of the system are weakly connected. Observe that the graphs PG_1 for state s_1 and PG_2 for state s_2 differ only by the new edge (u, v). Hence, action A is also enabled in u, and it may execute in the same way in s_2 as in s_1, which implies that u may call **exit**. This would disconnect v from the rest of the system and v remains disconnected from the system for the rest of the computation.

Hence, contrary to our initial assumption, $\mathcal{A}$ is not a self-stabilizing solution to the $\mathcal{FDP}$. A similar argument applies to the case in which node v or $v.C$ holds an identifier of u. □

So the problem is not solvable in the general case without the use of oracles.

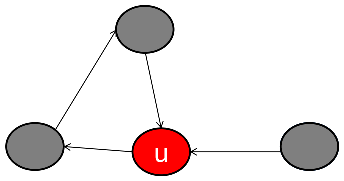


Figure 1.1: In this example node u cannot decide locally weather it is $safe$ to depart the system or not.

3.1 Oracles

To define the oracles we need to introduce some additional notation.

An edge (v, w) in PG with $v \neq w$ is *relevant* for some node u if $u = w$ and v is not gone, or it is implied by a message in $u.C$ carrying the id of w (i.e., $u = v$). Otherwise, the edge is *irrelevant* for u. Note that the edges implied by node ids stored in u are also irrelevant (meaning that u does not have to learn about them since it already knows them).

An oracle $\mathcal{O}$ is *id-sensitive* for some node u if its output depends on edges relevant for u. An oracle $\mathcal{O}$ is *strictly id-sensitive* if for every node u the oracle's output *only* depends on the edges relevant for u. Hence, the oracle ignores irrelevant edges. Note that an action that changes the system state without affecting relevant edges also does not affect the output of a strictly id-sensitive oracle. Naturally, a strictly id-sensitive oracle is also (regularly) id-sensitive. An oracle is *id-insensitive* if it is not id-sensitive. That is, the output of an id-insensitive oracle does not depend on the edges relevant for the node.

We define the following strictly id-sensitive oracles. Oracle $\mathcal{NID}$ (no identifiers) evaluates to **true** if the system does not contain an identifier of u in v or $v.C$ for some relevant node $v \neq u$. Oracle $\mathcal{EC}$ (empty channel) evaluates to **true** for a particular node u if the incoming channel of u is empty. Oracle $\mathcal{NIDEC}$ is a conjunction of $\mathcal{NID}$ and $\mathcal{EC}$. That is, $\mathcal{NIDEC}$ evaluates to **true** if both $\mathcal{NID}$ and $\mathcal{EC}$ evaluate to **true**. Note that $\mathcal{NIDEC}$ is less powerful than $\mathcal{NID}$ and $\mathcal{EC}$ used jointly since the algorithm using $\mathcal{NIDEC}$ is not able to differentiate between the conditions separately reported by $\mathcal{NID}$ and $\mathcal{EC}$. Oracle $\mathcal{SINGLE}$ evaluates to **true** for a node u if u shares edges with at most one relevant node.

Within a class of oracles $\mathcal{C}$, an oracle $\mathcal{O}$ is *necessary* for the $\mathcal{FDP}$ if for every algorithm $\mathcal{A}$ relying (i.e. needing for the solution) on an oracle $\mathcal{O}$'$\in\mathcal{C}$ with $\mathcal{O}'(s,u)$ =**true** while $\mathcal{O}(s,u)$ =**false** for some system state s and node u, $\mathcal{A}$ cannot be a self-stabilizing solution to the $\mathcal{FDP}$.

An Oracle $\mathcal{O}$ is *semi-persistent* if the actions of other nodes cannot invalidate it. That is, once a semi-persistent oracle is **true** for node u, it remains **true** regardless of actions of nodes other than u. Out of the oracles we defined, $\mathcal{NID}$ and $\mathcal{NIDEC}$ are semi-persistent while $\mathcal{EC}$ and $\mathcal{SINGLE}$ are not.

In this chapter, we show that without an id-sensitive oracle there is no self-stabilizing solution for the $\mathcal{FDP}$ within our model. Afterwards we show that among all id-sensitive oracles $\mathcal{SINGLE}$ is necessary to solve the $\mathcal{FDP}$. On the other hand, we prove that $\mathcal{NIDEC}$ is sufficient to solve the $\mathcal{FDP}$ by providing a self-stabilizing algorithm for the $\mathcal{FDP}$ relying on $\mathcal{NIDEC}$.

Problem $\mathcal{FSP}$, in contrast to the $\mathcal{FDP}$, does not require the nodes to irrevocably exit the system. This will allow us to design a self-stabilizing algorithm for the $\mathcal{FSP}$ that does not need any oracle.

4 Basic Properties of the $\mathcal{FDP}$

In this section we show that the $\mathcal{FDP}$ in the general case requires an id-sensitive oracle. Moreover, if only strictly id-sensitive oracles are considered, then $\mathcal{SINGLE}$ is necessary. The below proposition is a restatement of the results obtained in [34, 82]. Intuitively it says that once disconnected, the system may not be able to reconnect again.

Proposition 4.1 *[34, 82] If a computation of a copy-store-send algorithm starts in a state where two nodes u and v are disconnected in PG, u and v remain disconnected in PG in every state of this computation.*

Theorem 4.2 *Any self-stabilizing solution to the $\mathcal{FDP}$ has to rely on an id-sensitive oracle.*

Proof. In order to show this lemma it suffices to adapt and supplement the proof of Theorem 3.1. Assume that algorithm $\mathcal{A}$ is a self-stabilizing solution to the $\mathcal{FDP}$ that relies on an id-insensitive oracle $\mathcal{O}$. We consider following counterexample. Consider a system of at least three nodes. The computation of $\mathcal{A}$ starts in a state where all nodes but one, node v, are weakly connected. Hence, by Proposition 4.1, v remains disconnected from the system for the rest of the computation. Among the connected nodes, u is leaving. Since $\mathcal{A}$ is a solution to the $\mathcal{FDP}$, the computation will eventually reach a state s_1 in which u calls **exit** in some action A enabled in s_1.

We take s_1 and construct another state s_2 where there is a message carrying the id of v in the incoming channel of node u. In s_2, all nodes of the system are weakly connected. Observe that the graphs PG_1 for state s_1 and PG_2 for state s_2 differ only by the new, relevant edge (u, v). Since $\mathcal{O}$ is id-insensitive, both the state of u and the output of $\mathcal{O}$ for u are the same for s_1 and s_2. Hence, action A is also enabled in u, and it may execute in the same way in s_2 as in s_1, which implies that u may call **exit**. This would disconnect v from the rest of the system. By Proposition 4.1, v remains disconnected from the system for the rest of the computation.

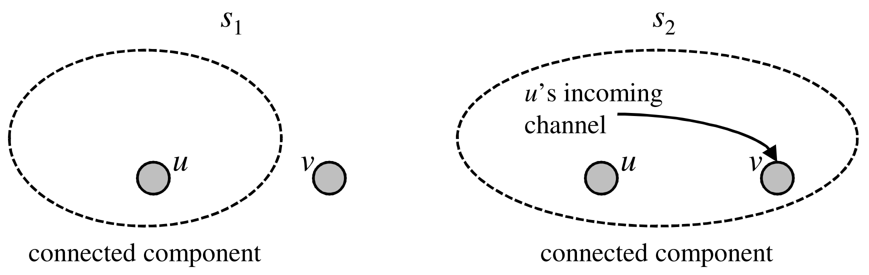


Figure 1.2: Illustration for the proof of Theorem 4.2.

Hence, contrary to our initial assumption, $\mathcal{A}$ is not a self-stabilizing solution to the $\mathcal{FDP}$. A similar argument applies to the case in which node v or $v.C$ holds an identifier of u. □

Theorem 4.2 immediately implies the following corollary.

Corollary 4.3 *A self-stabilizing solution to the $\mathcal{FDP}$ is impossible without an oracle.*

Interestingly, the impossibility even holds in a synchronous communication model. Consider the model in which each round consists of two stages: in stage 1, every node receives all messages from the previous round, and in stage 2, every node executes any number of its enabled actions. Let us transform the state s_1 in the proof of Theorem 4.2 into a state s_2 in which v has an edge to u. If this is the state of the initial round, u cannot receive a message from v in that round, since there was no prior round, so u still executes the **exit** statement. Hence, the system gets disconnected. We now address the strict id-sensitivity property of oracles.

Lemma 4.4 *If a self-stabilizing solution to the $\mathcal{FDP}$ relies on a strictly id-sensitive oracle, then this oracle evaluates to **true** only if a node has relevant edges with at most one relevant node.*

Proof. Assume there exists an algorithm $\mathcal{A}$ that is a self-stabilizing solution to the $\mathcal{FDP}$ which uses a strictly id-sensitive oracle $\mathcal{O}$ such that there exists a state s_1 where the oracle evaluates to **true** for some leaving node u while it shares relevant edges with at least two staying nodes v and w. That is, either u has an identifier of v or w in its incoming channel or u's identifier is in the memory of v or w or their respective incoming channels. We construct state s_2 by removing all edges from w except for the edges to u. Since $\mathcal{O}$ is strictly id-sensitive, this does not change the output of $\mathcal{O}$. Notice that in s_2, node w is disconnected from the system except for the edges to u.

Let us now consider a computation σ of $\mathcal{A}$ where u is leaving. Since $\mathcal{A}$ is a solution to the $\mathcal{FDP}$, u should eventually reach a state s_3 in σ in which it executes the **exit** statement in some enabled action A. Since A relies on $\mathcal{O}$, $\mathcal{O}$ must be true in this case.

We construct a system state s_4 where the state of u is the same as in s_3 while the state of the rest of the system is the same as in s_2. Since this does not change the edges relevant for u compared to s_2, this does not change the output of $\mathcal{O}$ compared to s_2. On the other hand, the local state of u and the output of $\mathcal{O}$ for u is the same in s_4 as in s_3. Hence, action A must be enabled in s_4, and it may execute in the same way in s_4 as in s_3, which implies that u may call **exit**. This, however, would disconnect node w from the rest of the staying nodes. According to Proposition 4.1, w remains disconnected from the system for the rest of the computation. Thus, contrary to the initial assumption, $\mathcal{A}$ is not a self-stabilizing solution to the $\mathcal{FDP}$. □

Lemma 4.4 leads to the following theorem.

Theorem 4.5 *Among all strictly id-sensitive oracles, the oracle $\mathcal{SINGLE}$ is necessary to obtain a self-stabilizing solution to the $\mathcal{FDP}$.*

We conjecture that $\mathcal{SINGLE}$ is also sufficient to solve the $\mathcal{FDP}$ but it is not semi-persistent. Semi-persistent oracles have the nice property that the action atomicity may be relaxed without affecting the result of the computation. It is known (cf., for example, [81]) that for oracle-free message-passing algorithms, for every low-atomicity computation (i.e., atomicity of action execution is only required within a node) there is an equivalent high-atomicity computation (i.e., only one action may be executed in the entire system at a time). In general, for a program with oracles this may no longer be true as the oracle may change its value in a low-atomicity computation due to the actions of other nodes. This can be particularly critical in an algorithm for the $\mathcal{FDP}$, where a node may think that it is safe to leave the network because its oracle evaluated to **true** but at the time when it calls **exit** the oracle may be **false** again. However, a semi-persistent oracle cannot be affected in that way. Actions of other nodes can only change its value from **false** to **true**. Hence the following proposition.

Proposition 4.6 *If an algorithm in an asynchronous message-passing system, whose successful performance relies only on the* **true***-value of oracles, uses semi-persistent oracles only, then for every low-atomicity computation, there is an equivalent high-atomicity computation.*

The proof is obvious and is omitted here. Since low-atomicity computation is what happens in practice, it is therefore preferable to find a solution relying on a semi-persistent oracle like $\mathcal{NIDEC}$.

5 Giving a First Solution

In this section we present a self-stabilizing algorithm called $\mathcal{SDA}$ that solves the Finite Departure Problem with the help of $\mathcal{NIDEC}$. We focus on the case that PG consists of a single connected component. However, the results transfer to PG being split up into multiple connected components. The algorithm is shown in Figure 1.3.

In algorithm $\mathcal{SDA}$, to maintain connectivity, each node p contains variables left and right that store node ids that are respectively less than and greater p. If left or right does not contain an identifier, it contains $-\infty$ or $+\infty$ respectively. To ensure a safe node departure, $\mathcal{SDA}$ uses the $\mathcal{NIDEC}$ oracle.

Algorithm $\mathcal{SDA}$ uses two message types: *intro* and *reverse*. A message of type *intro* carries a single node id and serves as a way to introduce nodes to one another. A message of type *reverse* does not carry an id. Instead, this message carries a boolean value denoted as **revright** or **revleft**. This message is a request for the recipient node to remove the respective left or right id from its memory and send its own id back.

We now describe the actions of the algorithm. Some of the actions contain the sending of messages involving ids stored in the left and right variables. If the variable contains $\pm\infty$, the sending action is skipped. To simplify the presentation of the algorithm, this is omitted in Figure 1.3.

The algorithm has three actions. The first action is *timeout*, which is a periodically triggered action, that has as a general goal to introduce the node to its neighbors. If the node is staying, it sends its id to its right and left neighbor. If the node is leaving, it sends messages to its neighbors requesting them to remove its id from their memory. If the node is leaving and the $\mathcal{NIDEC}$ oracle signals that it is safe to leave, the node introduces its neighbors to each other to preserve system connectivity and then exits by executing the **exit** statement. The second action is *introduce*. It receives and handles *intro* messages received by a node. The operation of this action depends on the relation between the id carried by the message and the ids stored in left and $right$. The node either forwards *intro*(id) to its left or right neighbor to handle it; or, if id happens to be closer to p than *left* or $right$, then p replaces the respective neighbor and instead introduces the old neighbor identifier to id. The third action, *reverse*, handles the neighbors' requests to leave, i.e. the *rev* messages received by a node. If p receives this message, it sets the respective variable to $+\infty$ or to $-\infty$ and, to preserve system connectivity, sends its own id to this node. To break symmetry, if p itself is leaving, it ignores the request from its left neighbor.

5.1 Correctness Proof

For $\mathcal{SDA}$ to be a self-stabilizing solution to the $\mathcal{FDP}$ it remains to show two properties. *Safety*: $\mathcal{SDA}$ never disconnects any relevant nodes. *Liveliness*: All leaving nodes eventually exit the system.

Lemma 5.1 *If a computation of $\mathcal{SDA}$ starts in a state where the graph PG of the non-gone nodes is weakly connected, the graph PG of the non-gone nodes remains weakly connected in every state of this computation.*

Proof. We demonstrate the correctness of the lemma by showing that none of the actions of $\mathcal{SDA}$ disconnects PG. Action *timeout* only adds edges to PG if $\mathcal{NIDEC}$ is **false** and cannot disconnect it in this case. If $\mathcal{NIDEC}$ is **true**, PG does not contain edges pointing to p and the only outgoing edges are $(p, left)$ and $(p, right)$. If p is connected to the rest of PG by at most one edge (i.e., $left$ or $right$ does not store an id), the departure does not disconnect PG. If both $left$ and $right$ store an id, the leaving of p does not disconnect PG because p sends $intro(left)$ to $right$ and $intro(right)$ to $left$ and thereby preserves weak connectivity between the remaining nodes.

Let us consider *introduce*. If the received id is the same as p or as *left* or $right$, the message is ignored. However, this does not disconnect PG. Let us consider the case of $id < p$. The case of $id > p$ is similar. There are two subcases to address. In case $id <$ *left*, p sends $intro(id)$ to *left*. That is, in PG, the edge (p, id) is replaced with (left, id). Since p stores the recipient identifier in *left*, i.e. PG has an edge (id, left), the graph connectivity is preserved. The other case is $\mathit{left} < id < p$. In this case, p replaces *left* with id and forwards the old value to id. That is, the edges (p, id) and (p, left) are replaced by (p, id) and (id, left). This replacement preserves PG connectivity.

The rev message received by a *reverse* action may force p to set either $right$ or *left* to infinity thus removing an edge from PG. Let us consider the case of $right$ being set to $+\infty$, the other case is similar. This operation removes $(p, right)$ from PG. However, *reverse* sends a message $intro(p)$ to $right$. That is, it replaces the edge $(p, right)$ with $(right, p)$, so weak connectivity of PG is preserved. □

The liveliness part of the correctness proof is more involved. Due to the way ids are handled by $\mathcal{SDA}$, the development of an edge can be traced over the course of the computation. Recall that an edge (p, q) is associated with an id of q stored in p or a message in $p.C$. The actions of $\mathcal{SDA}$ may transform (p, q) into a different edge (p', q'). Only the following cases can occur:

1. The *introduce* action stores q in *left* or $right$ or drops q since it is equal to p, *left* or $right$. In both cases, we stay with the edge (p, q).

```
constant    $p$ : node identifier
variables   $leaving$ : $boolean$, read only, true when $p$ wants to leave
            $left$ : node id less than $p$, $-\infty$ if undefined
            $right$ : node id greater than $p$, $+\infty$ if undefined
            $p.C$: channel of incoming messages of node $p$
messages    ($m.id$,intro): introduces node identifier
            $(direction, rev)$: requests recipient to reverse edge
                  $direction$ is revleft or revright

actions
timeout:    true $\longrightarrow$
              if not $leaving$ then
                send message($p, intro$) to $left$,
                send message($p, intro$) to $right$
              else  // leaving
                send message(revleft, $rev$) to $right$
                send message(revright, $rev$) to $left$
                if $\mathcal{NIDEC}$ then
                  if $left \neq -\infty$ and $right \neq +\infty$ then
                    send message($left, intro$) to $right$
                    send message($right, intro$) to $left$
                  exit
introduce:  $intro \in p.C \longrightarrow$
              receive $(m.id, intro)$
              if $m.id < left$ then
                send message($m.id, intro$) to $left$
              if $left < m.id < p$ then
                send message($left, intro$) to $m.id$
                $left := m.id$
              if $p < m.id < right$ then
                send message($right, intro$) to $m.id$
                $right := m.id$
              if $right < m.id$ then
                send message($m.id, intro$) to $right$
reverse:    $rev \in p.C \longrightarrow$
              receive $rev(direction)$
              if $direction$ = revleft then
                if not $leaving$ then
                  send message($p, intro$) to $left$
                  $left := -\infty$
              else  // $direction$ is revright
                send message($p, intro$) to $right$
                $right := +\infty$
```

Figure 1.3: Algorithm $\mathcal{SDA}$ for node p.

2. The *introduce* action may delegate the id of q to some node p': then (p, q) changes to (p', q). Note that whenever this happens, $p' \in [p, q]$.

3. The *reverse* action reverses the edge (p, q) to (q, p). Note that whenever this happens, p is staying or p is leaving and $p < q$.

The changes (i.e., cases 2 and 3) to an edge (p, q) over time form a sequence of edges $(p, q) = (p_0, q_0), (p_1, q_1), (p_2, q_2), \ldots$ that we call the *trace* of (p, q). The cases listed above imply the following lemma.

Lemma 5.2 *(Monotonicity) For every (p_i, q_i) in the trace of (p, q), $p_i, q_i \in [p, q]$.*

This and the fact that we have a finite number of nodes may seem to imply that every trace is finite, but for now we cannot exclude the case that an edge is reversed infinitely often between two nodes. It will only be implied later when we know that eventually all leaving nodes will exit the system.

Consider an arbitrary fixed computation of $\mathcal{SDA}$. An edge that does not change any more is called *stable*. A *steady chain* of nodes $x_k, \ldots, x_0$ is a sequence of leaving and not yet gone nodes of increasing order with stable edges (x_i, x_{i-1}). A steady chain is *maximal* if it cannot be extended to the left or right. Note that at every state of the computation, every leaving node is part of at least one maximal steady chain (which might just be a chain consisting of itself). Also, the following holds:

Lemma 5.3 *A maximal steady chain can only change in two ways: either (1) node x_k exits the system, or (2) the chain is extended to the left or right due to new stable edges.*

Since the number of nodes is finite, this means that eventually a maximal steady chain is stable, i.e., it does not change any more for the rest of the computation. We call this a *stable chain*. Now, we can prove the following lemma.

Lemma 5.4 *In every computation of $\mathcal{SDA}$, the only stable chain is the empty chain.*

Proof. Consider on the contrary that we have a non-empty stable chain $x_k, \ldots, x_0$. Our goal is to prove that eventually there is no incoming edge from non-gone nodes in PG to x_k.

First, suppose there is an incoming edge (p, x_k) with $p < x_k$. If there is a reversal in the trace of that edge, then we end up with a edge (x_k, p') with $p \leq p' < x_k$. If this causes x_k to delegate p' away, then due to Lemma 5.2 that edge will never include x_k again. Otherwise, x_k stores p' in *left*, and since a leaving node never reverses its edge to *left*, x_k either eventually delegates p' away, which will mean that the edge never includes x_k again, or x_k holds on to that edge, which means that (x_k, p') can never become an incoming edge to x_k again. So suppose that there is no reversal in the trace of (p, x_k). Then its trace is finite, which means that eventually it becomes a stable edge (p', x_k). We will argue via two cases that this cannot happen.

(1a) If p' is staying, then p' will eventually introduce itself to x_k. This will create a new edge (x_k, p') in PG. If this edge is not delegated by x_k, x_k will eventually ask p' to reverse its edge to x_k, which it will do, but that would contradict the assumption that (p', x_k) is stable. If x_k delegates (x_k, p'), then we keep track of that edge until we get to an edge (x, p') that gets reversed or is stable. In the former case, p' would delegate x_k to x, and in the latter case, p' would also either delegate x_k to x or reverse (p', x_k), depending on whether x is staying or leaving. Hence, in any case, (p', x_k) would not be stable, a contradiction.

(1b) If p' is leaving, then we distinguish between two cases. If x_k is not aware of p', then the chain can be extended to p' because (p', x_k) is stable, which contradicts our assumption to have a stable chain. If x_k is aware of p', then x_k will eventually ask p' to reverse its right edge, which will cause the edge (p', x_k) to be reversed, which again contradicts our assumption that (p', x_k) is stable.

Next, consider the case that there is an incoming edge (p, x_k) with $p > x_k$. If there is a reversal in the trace of that edge, we end up with an edge (x_k, p') with $x_k < p' \leq p$. If this causes x_k to delegate p' away, then due to the Lemma 5.2 the trace of that edge will never include x_k again. Otherwise, it must hold that $x_k < p' \leq x_{k-1}$. If $p' = x_{k-1}$, the edge would become stable, and otherwise, x_k would delegate x_{k-1} to p', which would contradict the assumption that (x_k, x_{k-1}) is stable. So in any case this edge will eventually not be an incoming edge to x_k any more. Thus, suppose that there is no reversal in the trace of (p, x_k). Then its trace is finite, which means that eventually it becomes a stable edge (p', x_k). We will again argue via two cases that this cannot happen.

(2a) If p' is staying, then p' will eventually introduce itself of x_k. If $x_k < p' < x_{k-1}$, then x_k would delegate x_{k-1} away, contradicting our assumption that (x_k, x_{k-1}) is stable. If $p' > x_{k-1}$, then similar arguments as for case (1a) above will show that (p', x_k) is not stable, also contradicting our assumption.

(2b) If p' is leaving, p' will eventually ask x_k to reverse its right edge, which it will do, contradicting our assumption that (x_k, x_{k-1}) is stable.

Moreover, x_k will never create an incoming edge to itself since it would only do that when asked to reverse (x_k, x_{k-1}), but since (x_k, x_{k-1}) is stable, this will not happen. Hence, eventually x_k has no incoming edge. This implies that eventually x_k has no more messages to process, so $\mathcal{NIDEC}$ will eventually be **true**. Therefore, x_k can exit the system, which contradicts our assumption that the chain is stable. □

Lemmas 5.1 and 5.4 lead to the following theorem.

Theorem 5.5 *Algorithm $\mathcal{SDA}$ and the $\mathcal{NIDEC}$ oracle provide a self-stabilizing solution to the $\mathcal{FDP}$.*

6 Relaxing the Problem

Until now we asked ourselves the question, whether a solution for the $\mathcal{FDP}$ is possible without using an oracle, and we have shown that unfortunately that is not the case. As we mentioned the graph classes where this holds is very limited.

However, avoiding the use of oracles is still desirable. So another approach besides looking into for which graph classes $\mathcal{FDP}$ is solvable without an oracle, is to relax the problem in such a way that it can be solvable for all graph classes. The idea is that we do not anymore require for the nodes to completely leave the system, but it suffices for all the leaving nodes to be in *sleep state*. More precisely, they should be in the *sleep state* and never wake up again, i.e *permanently asleep*. In that way the departures are simulated, since nodes which are *permanently asleep* do not participate any more in the system. That is how we come to introduce the Finite Sleep Problem.

***Finite Sleep Problem* ($\mathcal{FSP}$)** : eventually reach a legitimate state for the case that the **exit** command (and therefore the gone state) is *not* available (but only **sleep**).

Algorithm $\mathcal{SSA}$, which solves this problem, is almost identical to $\mathcal{SDA}$ shown in Figure 1.4. The only differences are that no oracle is checked and that the **sleep** command is used instead of **exit**.

```
constant     $p$ : node identifier
variables    $leaving$ : $boolean$, read only, true when $p$ wants to leave
             $left$ : node id less than $p$, $-\infty$ if undefined
             $right$ : node id greater than $p$, $+\infty$ if undefined
             $p.C$: channel of incoming messages of node $p$
messages     $(m.id, intro)$, introduces node identifier
             $(direction, rev)$, requests recipient to reverse edge
                       $direction$ is revleft or revright

actions
timeout:     true $\longrightarrow$
                 if not $leaving$ then
                     send message$(p, intro)$ to $left$,
                     send message$(p, intro)$ to $right$
                 else // leaving
                     send message(revleft, $rev$) to $right$
                     send message(revright, $rev$) to $left$
                     if $left \neq -\infty$ and $right \neq +\infty$ then
                         send message$(left, intro)$ to $right$
                         send message$(right, intro)$ to $left$
                     sleep
introduce:   $intro \in p.C \longrightarrow$
                 receive $(m.id, intro)$
                 if $m.id < left$ then
                     send message$(m.id, intro)$ to $left$
                 if $left < m.id < p$ then
                     send message$(left, intro)$ to $id$
                     $left := m.id$
                 if $p < m.id < right$ then
                     send message$(right, intro)$ to $m.id$
                     $right := m.id$
                 if $right < m.id$ then
                     send message$(m.id, intro)$ to $right$
reverse:     $rev \in p.C \longrightarrow$
                 receive $rev(direction)$
                 if $direction$ = revleft then
                     if not $leaving$ then
                         send message$(p, intro)$ to $left$
                         $left := -\infty$
                 else // $direction$ is revright
                     send message$(p, intro)$ to $right$
                     $right := +\infty$
```

Figure 1.4: Algorithm $\mathcal{SSA}$ for node p.

For the correctness proof of $\mathcal{SSA}$, we show that the safety and liveliness properties hold. We first define and prove the conditions that must prevail for a node to remain permanently asleep.

Lemma 6.1 *In the $\mathcal{SSA}$ algorithm, a node p is permanently asleep if and only if p is hibernating.*

Proof.

The backwards direction (if p is hibernating then p is permanently asleep) directly follows from Proposition 2.1. So it remains to prove the other direction.

Suppose that there is a node q that has a directed path to p along the nodes $q_0 = q, q_1, \ldots, q_\ell = p$ and q is either not asleep, or $q.C$ is non-empty. Without loss of generality, we may assume that for all other nodes q_i with $i \geq 1$, $q.C$ is empty (otherwise set q as the node with largest i with non-empty $q_i.C$). Hence, for all $i \geq 1$, q_{i+1} is initially stored in q_i. Since q is either awake and knows q_1, or $q.C$ contains a message with q_1, and q can only fall asleep in *timeout*, q is guaranteed to eventually process the edge (q, q_1) by either calling the *timeout* (which may contact q_1), *introduce* (which may contact or delegate q_1), or *reverse* action (which may contact q_1). If q_1 gets delegated, the receiving node is also guaranteed to process q_1. We continue the trace of (q, q_1) in this case (which causes the involved starting points to be woken up) until we reach a node q' where q_1 is not delegated any more. This must eventually happen since the number of nodes is finite. Hence, q_1 is eventually contacted, which will wake up q_1. Since q_1 initially stores q_2, q_1 is therefore also guaranteed to eventually process the edge (q_1, q_2). The same arguments as for q_1 then guarantee that also q_2 eventually processes the edge (q_2, q_3). Hence, by induction, eventually p is woken up, which completes the proof.

□

The lemma implies that given that our initial state satisfies the conditions in Section 1.2, no node will initially be permanently asleep. Additionally, the following lemma holds.

Lemma 6.2 *If a computation of $\mathcal{SDA}$ starts in a state where the graph PG of the non-hibernating nodes is weakly connected, the graph PG of the non-hibernating nodes remains weakly connected in every state of this computation.*

Proof.

We know from Lemma 5.1 that none of the actions of $\mathcal{SSA}$ disconnects the graph PG of the non-exited nodes. Thus, as long as no node falls asleep after an action (which can only happen if a leaving node calls *timeout*), the lemma holds. Suppose now that a leaving node p calls *timeout*. Our first goal is to show that no other node can become hibernating in this case. Consider any node $q \neq p$ that is non-hibernating and that has a directed path from p. We distinguish between two cases.

(1) If the directed path from p to q leads through a node q' stored in a message in $p.C$, then p cannot become hibernating and therefore q cannot become hibernating as well.

(2) If the directed path from p to q leads though $left$ or $right$ of p, then q cannot become hibernating because p will contact $left$ and $right$ in *timeout*.

Hence, if p does not become hibernating after *timeout*, weak connectivity is preserved. It remains to consider the case that p becomes hibernating. In this case, $p.C$ is empty. Also, there cannot be a path from a non-hibernating node q to p. Hence, $\mathcal{NIDEC}$ would be **true** for p (if evaluated by it). Since we know from Lemma 5.1 that in this case p may even exit the system without causing disconnection, we can also allow p to hibernate without risking disconnection for the non-hibernating nodes.

□

Lemmas 6.1 and 6.2 imply safety. So it remains to prove liveliness. Notice that if $\mathcal{NIDEC}$ is true (as a condition, not oracle) for a node p, then p would hibernate in $\mathcal{SSA}$ after calling *timeout*. Hence, it follows together with the proof of Lemma 6.2 that a node becomes hibernating in $\mathcal{SSA}$ if and only if $\mathcal{NIDEC}$ is true for it. On the other hand, a non-hibernating node is not permanently asleep. Therefore, the liveliness proof is identical to the liveliness proof of $\mathcal{SDA}$, which implies the following theorem.

Theorem 6.3 *$\mathcal{SSA}$ provides a self-stabilizing solution to the $\mathcal{FSP}$.*

7 Outlook

In this chapter, we showed that an id-sensitive oracle is required for a self-stabilizing solution to the Finite Departure Problem. We proved that among strictly id-sensitive oracles, $\mathcal{SINGLE}$ is necessary for a solution to the problem. We showed that a more restrictive oracle $\mathcal{NIDEC}$ is sufficient by presenting an algorithm that solves the Finite Departure Problem using $\mathcal{NIDEC}$.

Then we relaxed the Finite Departure Problem to the Finite Sleep Problem, in order to make it solvable without the use of an oracle.

It would be interesting to study the power of individual components of $\mathcal{NIDEC}$: $\mathcal{NID}$ and $\mathcal{EC}$. Specifically, we would like to determine the extent of the states from which the algorithm using only one of the components may recover.

Observe that the $\mathcal{SDA}$ algorithm (as also the $\mathcal{SSA}$), besides solving the $\mathcal{FDP}$, also organizes the staying nodes in a sorted list. It would be interesting to solve the $\mathcal{FDP}$ and $\mathcal{FSP}$ problems for other structures also. In the next chapter we take on this consideration.

Chapter 2

A General Framework for Dealing with Node Departures in Overlay Networks

1 Introduction

In the previous chapter we studied the handling of departures in overlay networks and introduced two problems: the *Finite Departure Problem ($\mathcal{FDP}$)* and the *Finite Sleep Problem ($\mathcal{FSP}$)*. In the $\mathcal{FDP}$, the leaving nodes have to irrevocably decide when it is safe to leave the network; whereas in the $\mathcal{FSP}$, this leaving decision does not have to be final: the nodes may resume computation when woken up by an incoming message. We showed that there is no self-stabilizing local-control protocol for the $\mathcal{FDP}$. But if an oracle is available, then an appropriate local-control protocol can be constructed. Moreover, a variant of that protocol can solve the $\mathcal{FSP}$ without using an oracle. However, these protocols require that there is a fixed total order on the nodes (e.g., their names or IP addresses do not change), and they only work for a specific overlay maintenance protocol that aims at organizing the nodes in a sorted list.

In this chapter, we present a self-stabilizing protocol for the $\mathcal{FDP}$ that can extend a large class of overlay maintenance protocols so that they are then guaranteed to eventually exclude the leaving nodes without risking disconnection and while the overlay maintenance protocol is operating as specified for the staying nodes. As a by-product, we present a set of four basic primitives for the manipulation of edges in overlay networks that are safe and universal in a sense that connectivity is preserved and that, in principle, one can get from any weakly connected graph to any other weakly connected graph. This might be of independent interest as we expect our insights to simplify the design and analysis of overlay maintenance protocols in the future.

1.1 Model and Preliminaries

A node u has a variable *u.mode* $\in$ {leaving, staying} that is read-only. If this variable is set to **leaving**, the node is *leaving*; the node is *staying* if the variable is set to **staying**.

1.2 Our Results

Our main result is a self-stabilizing local-control protocol presented in Section 3 that can solve the $\mathcal{FDP}$ when relying on the $\mathcal{SINGLE}$ oracle. Compared to the protocols in the previous chapter, this protocol has the nice property that no fixed order on the nodes is needed. So an underlying layer may change referencing information about a node like its IP address, web address, port number, or pseudonym as it likes. The only interface that our protocol needs to that layer is that it can send a message to a node identified by some reference or by executing $v = w$ or $v \neq w$ to check whether two references point to the same or different nodes. That does not just allow the actions of the underlying network layer to be decoupled from the application layer but it also simplifies using our departure protocol together with other overlay maintenance protocols, as we will demonstrate in this chapter. In order to simplify the analysis and formally specify the class of overlay maintenance protocols that can be used in conjunction with our departure protocol, we point out that the solutions in this chapter require the additional constraint that initially there exists at least one staying node per connected component of the overlay network.

2 The Basic Algorithm

Our protocol consists of various actions. In the $present$ action, a reference v is introduced to some node (i.e., whenever it is called for some v, the calling node performs the Introduction primitive for v). Instead, in the $forward$ action, a reference v is delegated to some node.

We assume that whenever a node a sends a request to call $present$ of $forward$ containing a reference of a node b to another node c, it automatically sends some relevant information it knows about b along with it. In this section the only relevant information is the *mode* of b, which we denote as $a.(b.mode)$ (i.e. a's knowledge of b's mode), which can be **staying** or **leaving**. Note that $a.(b.mode)$ might be incorrect (i.e., $a.(b.mode) \neq mode(b)$) since b might have a different mode than a thinks it has.

We denote the set of all references a node u stores in its local memory as the neighborhood set $u.N$ of u. Note that $u.N$ is not a variable of u but just a notation we use, which simplifies our protocol description and the proofs. Along with each $v \in u.N$, node u also stores its knowledge of the mode of v, denoted by $u.(v.mode)$. Our solution makes use of a special variable called $anchor$ whose reference is the only one *not* being in $u.N$. $anchor$ will only be used by the leaving nodes, so in a legitimate state, the $anchor$ of a staying node is empty, denoted by $\perp$. The anchor is a reference of a node which a leaving node v assumes to be staying. Therefore, each time v gets a message from a third node w, v forwards w to its anchor by a $forward$ message in the hope of eliminating all references to itself. Each node has a periodically executed $timeout$ action. In case a node is leaving, it sends a $present$ message to its anchor in the $timeout$ action (in order to verify it has a staying anchor). If it is staying, it sends a $present$ message containing its own reference to all neighbors (to make other nodes aware of it). This is an implementation of our earlier presented *self-introduction* primitive. Periodically executed self-introduction can ensure that invalid information vanishes from the system, as we will show later. Additionally, leaving nodes consult $\mathcal{SINGLE}$ in $timeout$, and if it evaluates to true, the node is safe to perform **exit**. The actions of our protocol are presented in Algorithms 1- 3.

2.1 Correctness Proof

To show that our proposed protocol is a self-stabilizing solution to the $\mathcal{FDP}$, it remains to show two properties.

***Safety*:** The protocol never disconnects any relevant nodes.

Algorithm 1 $u.timeout$

```
1: if $u.anchor \neq \perp$ and $u.(anchor.mode) = leaving$ then
2:     send message(u.anchor,present) to $u$                          $\triangleright \diamondsuit$
3:     $u.anchor \leftarrow \perp$
4: if $u.mode = leaving$ then
5:     if $u.N = \emptyset$ then
6:         if $\mathcal{SINGLE}$ then
7:             exit
8:         else
9:             if $u.anchor \neq \perp$ then
10:                send message(u,present) to $u.anchor$               $\triangleright \diamondsuit$
11:    else
12:        for all $v \in u.N$ do
13:            send message(v,forward) to $u$                          $\triangleright \diamondsuit$
14:        $u.N := \emptyset$
15: else
16:    if $u.anchor \neq \perp$ then
17:        send message(u.anchor,present) to $u$                       $\triangleright \diamondsuit$
18:        $u.anchor \leftarrow \perp$
19:    for all $v \in u.N$ do
20:        if $u.(v.mode) = leaving$ then
21:            $u.N := u.N \setminus \{v\}$
22:        send message(u,present) to $v$                              $\triangleright \diamondsuit$ or $\clubsuit$
```

Algorithm 2 message(v,present) $\in u.C$

```
1: if $v = u.anchor$ and $u.(v.mode) = leaving$ then
2:     $u.anchor \leftarrow \perp$                                     $\triangleright \spadesuit$
3: if $u.(v.mode) = leaving$ then
4:     if $u.mode = leaving$ then
5:         send message(u,forward) to $v$                              $\triangleright \clubsuit$
6:     else
7:         if $v \in u.N$ then
8:             $u.N := u.N \setminus \{v\}$
9:         send message(u,forward) to $v$                              $\triangleright \clubsuit$
10: else
11:    if $u.mode = leaving$ then
12:        if $u.anchor \neq \perp$ then
13:            send message(u,forward) to $v$                          $\triangleright \clubsuit$
14:        else
15:            $u.anchor := v$
16:    else
17:        $u.N := u.N \cup \{v\}$                                     $\triangleright \spadesuit$
```

Algorithm 3 message(v,forward) $\in u.C$

```
1: if $v = u.anchor$ and $u.(v.mode) = leaving$ then
2:     $anchor \leftarrow \perp$                                    ▷ ♠
3: if $u.(v.mode) = leaving$ then
4:     if $u.mode = leaving$ then
5:         if $u.anchor = \perp$ then
6:             send message(u,forward) to $v$                      ▷ ♣
7:         else
8:             send message(v,forward) to $u.anchor$               ▷ ♡
9:     else
10:        if $v \in u.N$ then
11:            $u.N := u.N \setminus \{v\}$
12:        send message(u,forward) to $v$                          ▷ ♣
13: else
14:    if $u.mode = leaving$ then
15:        if $u.anchor \neq \perp$ then
16:            send message(v,forward) to $u.anchor$               ▷ ♡
17:        else
18:            $u.anchor := v$
19:    else
20:        $u.N := u.N \cup \{v\}$                                  ▷ ♠
```

***Liveness*:** All leaving nodes are eventually gone.

Lemma 2.1 *If a computation of our protocol starts in a state where the sub-graph PG of relevant nodes is weakly connected, it remains weakly connected in every state of the computation.*

To prove safety we make use of the results from the prologue.

Proof. First of all, note that each relevant node is also awake, since obviously gone nodes cannot be relevant. The proof of the lemma relies on the fact that our protocol that the (awake) nodes run is a composition of the four primitives presented in the prologue. To illustrate this, the protocol is annotated with the symbols ♢, ♡, ♠, ♣. Each symbol represents a primitive: ♢ is (Self-)Introduction, ♡ is Delegation, ♠ is Fusion and ♣ is Reversal. Therefore, we can use the result of Lemma 3.1 and the fact that $\mathcal{SINGLE}$ preserves weak connectivity in the only case in which we do not use a primitive, (i.e., a node executes **exit**). This proves the lemma. □

It remains to show that our protocol makes progress such that all leaving nodes eventually leave the system.

Theorem 2.2 *Leaving nodes eventually execute the* ***exit*** *command, thereby preserving liveness.*

Proof.

The first step to prove liveness is to show that eventually all information in the system is valid.

Lemma 2.3 *During the execution of the protocol, the system eventually reaches a valid state.*

Proof. Let Φ_t be a potential function that denotes the amount of invalid information present in the system at some time t, i.e., Φ_t is equal to the number of edges (x, y), either explicit or implicit, such that $y.mode \neq x.(y.mode)$.

At first, note that no action conducted by any node can increase Φ_t. That is because the only way Φ_t can increase is if invalid information is copied. In order to do so, a node u has to forward invalid information about node v to another node w , since the information sent about oneself is always valid. The only spots in the pseudocode where this can potentially happen are lines 8 and 16 of the $forward$ action, where u sends $(v, forward)$ to its anchor. However, in that case v is not saved by u. Thus , even if u sends invalid information about v to its anchor, the invalid information is not duplicated in the system. Therefore, $\Phi_t \geq \Phi_{t'}$ for any $t' > t$. To conclude the lemma, it suffices to show that as long as $\Phi_t > 0$ it holds that for any t there is a $t' > t$ such that $\Phi_{t'} < \Phi_t$.

First of all, note that whenever a node learns that its anchor is leaving, it immediately changes the variable $u.anchor$ to $\perp$. Let (u, v) be an edge that contains invalid information at time t. We consider the following four cases.

1. $v.mode = leaving, u.(v.mode) = staying, u.mode = leaving.$

   We can distinguish between two subcases.

   (a) In the first sub-case we consider that u has stored v in its local memory (i.e. $u.N$ or $u.anchor$). If v is $u.anchor$ then at some point u calls the $timeout$ action and sends $(u, present)$ to v (or executes **exit** in case the oracle evaluates to true and the invalid information vanishes). Eventually, v receives the $(u, present)$ message and answers with a $(v, forward)$ message to u, so u gets valid information about v. Note that at no point u propagates the invalid information about v. On the other hand, if $v \in u.N$, then u sends $(v, forward)$ to itself in the $timeout$ action and as a consequence u sends $(u, forward)$ to v. Consequently, u no longer maintains invalid information about v.

   (b) In the second sub-case v is contained in a message in $u.C$. We assume that u has another anchor a. Otherwise, after handling the message, u would set its anchor to v, thereby saving v in its local memory and this situation is discussed in the previous sub-case. We also assume that $a.mode = staying$, since otherwise the edge (u, a) would fit to the first sub-case and $\Phi_{t'}$ would decrease anyway compared to Φ_t. If $(v, present)$ is in $u.C$, $(u, forward)$ is sent back to v and the invalid edge vanishes. In case $(v, forward)$ is in $u.C$, then according to the protocol u forwards v to a, thereby pushing the invalid information to a. The anchor a eventually stores v in $a.N$ and during its $timeout$ action it sends $(a, present)$ to v. When v receives the $(a, present)$ message, it either sends a $(v, forward)$ message back in case it already has an anchor, or, if it has no anchor, it sets $v.anchor = a$ and eventually sends $(v, present)$ to a during the $timeout$ action. One way or another, a learns the valid information about v and deletes v from $a.N$.

   So all in all we have one less edge with invalid information.

2. $v.mode = leaving, u.(v.mode) = staying, u.mode = staying.$

   If $(v, present)/(v, forward) \in u.C$, then u includes v in $u.N$ in its $timeout$ action, so we only need to consider the case in which $v \in u.N$. So if $v \in u.N$, u sends $(u, present)$ to v during the $timeout$ action. Node v either sends $(v, forward)$ back to u if it has an anchor, or it sets u as its anchor and sends $(v, present)$ back to u during the $timeout$ action. Either way u learns the valid information about v, v is deleted form $u.N$ and the amount of invalid information decreases.

3. $v.mode = staying$, $u.(v.mode) = leaving$, $u.mode = leaving$.

   If $v = u.anchor$, u sets $u.anchor$ to $\perp$ during $timeout$ and sends a $(v, present)$ message to itself. Additionally, if $v \in u.N$, u sends $(v, forward)$ to itself in $timeout$, therefore we only need to consider the situations in which $(v, present)/(v, forward) \in u.C$. In this case, u either sends $(u, forward)$ to v or $(v, forward)$ to its anchor a. In the first sub-case u no longer maintains invalid information about v. For the second sub-case we assume that $a.mode = staying$, because otherwise we are back to case 1 for edge (u, a) (and can show that $\Phi_{t'}$ decreases). As u forwards the invalid information to a, a in its turn sends $(a, forward)$ to v and deletes v from $a.N$. So the invalid edge (a, v) vanishes.

4. $v.mode = staying$, $u.(v.mode) = leaving$, $u.mode = staying$.

   If $v \in u.N$, u sends $(v, forward)$ to itself in $timeout$ and again we only need to consider situations in which $(v, present)/(v, forward) \in u.C$. In this case u sends $(u, forward)$ to v and the invalid edge vanishes.

Therefore, as long as $\Phi_t > 0$ for any t there is a $t' > t$ such that $\Phi_{t'} < \Phi_t$ and eventually all invalid information vanishes. □

Now we are able to prove Theorem 2.2. At first note that at this point we can assume that the mode information in the whole network is valid. Secondly, as long as there are leaving nodes that are not yet gone and as long as the network does not consist entirely of leaving nodes, at least one leaving node has at least one edge with a staying node. Therefore, it suffices to show that eventually a leaving node which has an edge to or from some staying node u executes **exit**. By using this argument inductively we have that eventually all leaving nodes execute **exit**.

Let v be a leaving node that has at least one explicit or implicit edge to a staying node u. By definition of PG this means that either (i) v has a message in $v.C$ carrying a reference of u or (ii) vice-versa or (iii) v stores a reference of u in its local memory or (iv) vice-versa.

Note that in all four situations v eventually gains an anchor. If v has another anchor than u, this holds trivially. Otherwise, in case (i), v eventually receives the message in $v.C$ and then sets its anchor to u. In case (ii), u receives the message in $u.C$ and responds with a $(u, forward)$ message to v, which results in case (i). In case (iii) either u is already the anchor or v sends a $(u, forward)$ message to itself in $timeout$, which is again case (i). In (iv) u sends a $(u, present)$ message to v in $timeout$, which is again case (i). Therefore, every leaving node that has an edge to a staying node, eventually stores a valid anchor.

From the actions of our protocol we know that every node reference that is either in the local memory of v or in a message in $v.C$ is sent to $v.anchor$ in the $timeout$ action and the $present/forward$ actions respectively. The only nodes that prevent $\mathcal{SINGLE}$ from evaluating to **true** are those which either store a reference v in their local memory or have a reference to v in a message in their channel, but not vice-versa. Let x be such a node. We now show that eventually x does not prevent $\mathcal{SINGLE}$ from evaluating to **true** for v. We consider all possible subcases: x can be staying or leaving and the reference to v can be in a message or in the local memory.

At first we consider the case that x is staying and has v in its local memory. According to our protocol x sends $(x, present)$ to v in its $timeout$ action. Once v receives this message it sends

$(v, forward)$ back to x (because it already has an anchor). This delegation forces x to delete v from its local memory and to send another $(x, forward)$ back to v. Now x does not have an edge to v anymore and x cannot prevent $\mathcal{SINGLE}$ from evaluating to **true**. In case x is staying and has a message containing v in $x.C$, x does not save v in its local memory, independent of the fact whether the message is a $(v, present)$ or $(v, forward)$ message. It sends a $(x, forward)$ message back to v, such that the implicit edge (x, v) is deleted from the PG.

In case x is leaving and has v in its local memory, x creates a $(v, forward)$ message, deletes the reference to v from the local memory and sends the $(x, forward)$ message to itself in $timeout$. This leads us directly to the last case in which x is leaving and has a message containing v in $x.C$. If the message is a $(v, present)$ message, x sends a $(x, forward)$ message back to v (which v forwards to its anchor). If the message is a $(v, forward)$ message, either $(x, forward)$ is sent to v or a $(x.anchor, present)$ message is sent to v (which v also forwards to its anchor). Again in both cases the implicit edge (x, v) is deleted from the PG.

Thus, we can conclude that as soon as v has a valid anchor, eventually all leaving nodes and all staying nodes (except for the anchor) do not have an edge to v. So $\mathcal{SINGLE}$ can evaluate to **true** and v can execute **exit**.

Consequently, a leaving node that is not gone yet and has an edge to an awake node, eventually has a valid anchor and is ultimately able to execute **exit**. As long as there are not gone leaving nodes, there exists such a node, which implies that in finite time all nodes eventually execute **exit**. □

3 Embedding in Existing Overlay Protocols

In this section we show how the protocol that was developed in Section 2 can be embedded in any existing distributed overlay protocol. With this general embedding approach, existing overlay management protocols can be easily modified in order to safely deal with node departures, which stresses the adaptability of the presented approach. Note that the original protocol does not necessarily have to be self-stabilizing. However, it must work correctly, i.e., not disconnect the overlay topology through its actions.

To proceed with this chapter we introduce the notion of *safe introduction*, i.e., a way for a node u to send a message m to a node v, such that invalid information is not duplicated and eventually vanishes, regardless of the structure of the message m. The general idea behind the concept of safe introduction is to reinforce a message sent from u to v by a number of elementary $present$ messages sent to the references included in the original message, once v receives the message. In that way the nodes whose references are contained in m introduce themselves directly to v, in order to avoid propagation of invalid information about themselves. We show that by using *safe introduction* the commands executed by a node can be decomposed into the four primitives and thereby preserve connectivity. Also when used in combination with the *self introduction*, we can show that invalid information eventually vanishes. More specifically, safe introduction guarantees that invalid information is not duplicated, whereas self introduction ensures that the valid (correct) information is periodically forwarded such that invalid information eventually vanishes.

The framework given below is based on these two procedures and, as we show, solves the $\mathcal{FDP}$ or the $\mathcal{FSP}$ respectively. The exact details are explained in the following sub-section.

3.1 FDP for Arbitrary Protocols

In order to embed our protocol into an existing overlay protocol $P \in \mathcal{P}$, the only algorithmic requirement P must fulfill is to conduct periodic *self introduction*, i.e., to have a periodically executed $timeout$ action, in which a node sends a message with information about itself (reference and mode) to every node in its neighborhood. Therefore, the $timeout$ action for a node u must contain a command of the form $\forall v \in u.N$ send message(u,present) to v. Note that this does not mean that u has to be the only argument contained in this self-introduction messages of P. If there are multiple arguments, our approach described below works as desired. However, for simplification, each node u also sends $(u, present)$ message with u being the only parameter to each node in $u.N$. This introduces some minor message overhead (in a worst-case scenario u sends every message twice), but since we are not concerned with message complexity, this is just a minor issue. To avoid renaming issues, we assume that P does not already use the names $process$ and $forward$ for its actions, such that those message type (and action) names are exclusively introduced by the new protocol.

The idea of the general framework is to replace each command in P, in which some node sends a message to another node, with a series of message sending commands. This ensures that weak connectivity is not lost and node departures are possible. More specifically, in order to construct the new protocol P', we do as follows: All message sending commands in P where a node u forwards some node references $x_1, \ldots, x_k$ to a node v are also reinforced by requests for $present$ messages, in order for message information to be received correctly and the corresponding action to be carried out.

We construct P' by replacing every message of the form "send message($x_1, \ldots, x_k$,m.type) to v" of P with a message "send message(m.type,[$x_1, \ldots, x_k$],process) to v" The $process$ message has the purpose to inform v about the $m.type$ message. Once the $process$ message is received by a node v, v contacts the nodes referenced in that message $(x_1, \ldots, x_k)$ by sending $present$ messages to them, in order to request and receive the valid information (in the current case the valid $mode$ information) about them. In this specific step we implicitly use the above mentioned idea of safe introduction. We make sure that $m.sender$ can not spread invalid information about the x_i nodes to v.

Another addition is that every node u executing P' is required to maintain an additional variable $u.anchor$, which (in a valid configuration) has the value $\perp$ if u is staying. Moreover, we need a variable called $u.mlist$ for every node u which stores all the $m.type$ commands u received by $process$ messages. These are executed once the valid information from the x_i nodes arrive by $(x_i, forward)$ messages. For each node v stored in $u.mlist$ an additional value $u.(v.certainty)$ is stored, which can have the values $true$ or $false$, indicating to whether the mode information about v came from v itself or not. We assume that a node stored in $u.mlist$ is automatically in $u.N$, i.e., $u.mlist \subseteq u.N$. Remember that the $u.anchor$ variable is not part of $u.N$.

The framework for constructing the modified protocol P' is given here. Note that instead of writing $sendmessage(v, \perp, \perp, forward)$ and $sendmessage(v, \perp, \perp, present)$ if the last two parameters are not needed (i.e., we make use of the messages presented in the last section), we stick to the notation of $sendmessage(v, forward)$ and $sendmessage(v, present)$.

Correctness Proof

We need to show that protocol P' eventually works like P (e.g., reaches the same target topology) and that all leaving nodes are eventually gone.

Algorithm 4 $u.timeout$

```
1: if $u.anchor \neq \perp$ and $u.(anchor.mode) = leaving$ then
2:     send message(u.anchor,present) to $u$                                   $\triangleright \diamondsuit$
3:     $u.anchor \leftarrow \perp$
4: if $u.mode = leaving$ then
5:     see commands of Algorithm 1.
6: else
7:     if $u.anchor \neq \perp$ then
8:         send message(u.anchor,present) to $u$                               $\triangleright \diamondsuit$
9:         $u.anchor \leftarrow \perp$
10:    Execute $P - timeout$ action of $P$ but:
11:    for each [send message($x_1, \dots, x_k$,m.type) to $v$] $\in$ timeout(P) do
12:        send message(m.type,[$x_1, \dots, x_k$],process) to $v$            $\triangleright \diamondsuit/\heartsuit$
13:    for all $v'$ in entries $m$ in $u.mlist$ do
14:        if $u.(v'.mode) = unknown$ then
15:            send message(u,present) to $v'$                                $\triangleright \diamondsuit/\heartsuit$
16:    for all $v \in u.N$ do
17:        if $u.(v.mode) = leaving$ and $v \notin u.mlist$ then
18:            send message(u,present) to $v$                                 $\triangleright \clubsuit$
19:            $u.N := u.N \setminus \{v\}$
```

Algorithm 5 (m.type,[$x_1, \dots, x_k$],process) $\in u.C$

```
1: if $u.mode = staying$ then
2:     for all $x_i$ do
3:         $u.(x_i.mode) := unknown$
4:     Add $[m.type, [x_i, \dots, x_k]]$ to $u.mlist$                          $\triangleright (\spadesuit)$
5: for all $x_i$ do
6:     send message(u,present) to $x_i$                                        $\triangleright \diamondsuit$
```

Algorithm 6 message(v,present) $\in u.C$

```
1: if $v = u.anchor$ and $u.(v.mode) = leaving$ then
2:     $u.anchor \leftarrow \perp$
3: if $u.(v.mode) = leaving$ then
4:     if $u.mode = staying$ then
5:         if $v \in u.N$ then
6:             $u.N := u.N \setminus \{v\}$
7: else
8:     if $u.mode = leaving$ then
9:         if $u.anchor = \perp$ then
10:            $u.anchor := v$
11: send message(u,forward) to $v$                                             $\triangleright \clubsuit$
```

Algorithm 7 (v,forward) $\in u.C$

```
if $v = u.anchor$ and $u.(v.mode) = leaving$ then
    $anchor \leftarrow \perp$
if $u.(v.mode) = leaving$ then
    See commands of Algorithm 3
else
    if $u.mode = leaving$ then
        if $u.anchor \neq \perp$ then
            send message(v,forward) to $u.anchor$                       $\triangleright \heartsuit$
        else
            $u.anchor := v$
    else
        $u.N := u.N \cup \{v\}$                                         $\triangleright \spadesuit$
if $u.mode = staying$ then
    for each $[m.type[x_1, \dots, v, \dots, x_k]] \in u.mlist$ do
        $u.(x_i.mode) := u.(v.mode)$
        if $\forall i, u.(x_i.mode) \neq unknown$ then
            if $\forall i, u.(x_i.mode) = staying$ then
                execute $u.(m.type(x_1, \dots, x_k))$ as in $P$
            else
                for all $v' \in [x_1, \dots, x_k]$ do
                    send message(u,forward) to $v'$                     $\triangleright \clubsuit$
                    $u.N := u.N \setminus \{v'\}$
                delete $[m.type[x_1, \dots, x_k]]$ from $u.mlist$
        else
            for all $v' \in [x_1, \dots, x_k]$ do
                send message(u,present) to $v'$                        $\triangleright \diamondsuit$
```

Theorem 3.1 *Let $P \in \mathcal{P}$ be a distributed overlay protocol which solves some distributed problem $\mathcal{DP}$ and contains a $timeout$ action in the form $\forall v \in neighborhood(u)$ send message(u,present) to v. Then there is another protocol P' constructed as described above, such that P' eventually solves the $\mathcal{FDP}$ as well as $\mathcal{DP}$.*

Proof. At first we show that P' still fulfills the safety condition.

Lemma 3.2 *If a computation of the protocol P' starts in a state where the sub-graph PG of the relevant nodes is weakly connected, it remains weakly connected in every state of the computation.*

Proof. The actions migrated from P preserve connectivity since $P \in \mathcal{P}$. The rest of the actions are adaptions of the protocol of Section 2 (for which we showed in Lemma 2.1 that they preserve connectivity), with actions that are almost alike. In fact, all the newly introduced commands can be decomposed to the four primitives presented in the prologue (see annotations on the algorithms in the pseudocode) Therefore, weak connectivity is preserved. □

Next we show that eventually we have only valid information in the system (Lemma 3.4 and 3.3).

Lemma 3.3 *During the execution of P' eventually there is no invalid mode information in the system (i.e., invalid information regarding the mode variables, all nodes save about other nodes).*

Proof. Let Φ_t be a potential function that denotes the amount of invalid mode information present in the system at some time t, i.e., Φ_t is equal to the number of edges (x, y), either explicit or implicit, such that $mode(y) \neq x.(y.mode)$.

Again, note that no action conducted by any node can increase Φ_t *in the long run*. The only way Φ_t can increase is if invalid information is copied. In order to do so, a node u has to forward invalid information about node v to another node w , since the information sent about oneself is always valid. This could happen is in the $forward$ action, at line 8 of the pseudocode, where u sends $(v, forward)$ to $u.anchor$. However, as shown in the proof of Lemma 2.3, this is not the case. The other possibility where information about a third node is propagated is in the $timeout$ action at line 15. In line 14 a process message is sent to v with mode information of nodes $x_1, \ldots, x_k$. However, once v receives this message the mode information is stored in v with $v.(x_i.certainty) = false$, and the protocol forces v to contact the x_is to get their mode information. The $m.type$ action sent within the process message is only executed if all responses from these nodes in form of a $(x_i, forward)$ messages arrive. Thereby, v learns the valid information about these nodes and eventually the invalid information about the x_i vanishes. Therefore, even though the amount of invalid information at first can increase, it is corrected eventually.

Note that, if $\Phi_t = 0$ then $\forall t' > t : \Phi_{t'} = 0$. To conclude the lemma, it suffices to show that as long as $\Phi_t > 0$ there is a $t' > t$ such that $\Phi_{t'} < \Phi_t$. First of all, note that whenever a node learns that its anchor is leaving, it immediately changes the variable $u.anchor$ to $\perp$. Let (u, v) be an edge that contains invalid mode information at time t (i.e., u has invalid information about v's mode).

We consider the following four cases.

1. $v.mode = leaving$, $u.(v.mode) = staying$, $u.mode = leaving$:

   We can distinguish between two subcases.

(a) In the first sub-case u has stored v in $u.N \cup u.anchor$. If v is $u.anchor$ then at some point u calls the $timeout$ action and eventually sends $(u, present)$ to the anchor v (or it executes **exit** before that and the invalid information is not present anymore). So eventually v receives a $(u, present)$ message and in its turn sends a $(v, forward)$ message to u, so u eventually gets valid information for v and deletes its anchor. Note that at no point u propagates the invalid information about v. On the other hand, if $v \in u.N$, then u sends $(v, forward)$ to itself, and as a consequence u sends $(u, forward)$ to v. Node u no longer has information about v and the invalid mode information is lost.

(b) In the second sub-case v is contained in a message in $u.C$. We can assume that u has another anchor a. Otherwise, after the handling the message, u would set its anchor to v, thereby saving v in its local memory, which is handled in the previous sub-case. We also assume that $a.mode = staying$, since otherwise, the edge (u, a) would fit again in the first sub-case. If the message is a $present/process$ message, $(u, forward)$ is sent back to v and the invalid edge vanishes, since v is not stored by u. If the message is a $(v, forward)$ message, then according to the protocol u sends $(v, forward)$ to a. Anchor a stores v in $a.N$, and during its $timeout$ action send $(a, present)$ to v. When v receives the $(a, present)$ message and it has already an anchor it eventually sends a $(v, forward)$ message back. If it has no anchor it sets $v.anchor = a$ and eventually sends $(v, present)$ to a during the $timeout$ action. Either way a learns valid information about v.

So all in all we have one less edge with invalid information.

2. $v.mode = leaving$, $u.(v.mode) = staying$, $u.mode = staying$:

   Again, either $v \in u.N$ or a message containing the reference of v is in $u.C$.

   (a) We consider first that $v \in u.N$. It is possible that a command of the form [m.type($x_1, \ldots, x_k$)] is activated by some $forward$ message. This command could include the sending of various messages of the form [u: send message($\ldots, v, \ldots$,m.type') to z] in P.

   In that case this command is translated in P' to a $process$ message, but z stores v as uncertain (i.e. $z.(v.certainty) = false$) if z is staying (if z is leaving we are back at case 1) and eventually receives a $(v, forward)$ message from v itself (by construction of our protocol). Therefore, z eventually learns valid information about v.

   In any case, u eventually sends a $(v, present)$ message to v during the $timeout$ action. Node v either sends $(v, forward)$ back to u if it v has no anchor, or sets u as its anchor and sends $(v, present)$ back to u during the $timeout$ action. Eventually u learns valid information about v.

   (b) Now we consider the case that a message containing the reference of v is $\in u.C$. In case a the message is a $(v, present)$ or $(v, forward)$ message, u includes $v \in u.N$ and sends $(u, present)$ back to v in its $timeout$ action, which is the situation handled in the first sub-case. Moreover, if $(v, forward) \in u.C$, it can be that a $m.type$ action is triggered, which is also handled in the first sub-case. In case $(\ldots, v, \ldots, process) \in u.C$ we are also in the first sub-case, since this specific situation is considered there.

3. $v.mode = staying$, $u.(v.mode) = leaving$, $u.mode = leaving$:

If $v = u.anchor$, u sets its $u.anchor$ to $\perp$ in $timeout$ and sends $(v, present)$ to itself. Also, if $v \in u.N$, u deletes v from $u.N$ and sends $(v, forward)$ to itself in $timeout$ (or executes $exit$ before that, so the invalid information is not present anymore) so we only need to consider the case in which a message containing the reference of v is $\in u.C$. In this case, u either sends $(u, forward)$ to v or v to its anchor a. In the first sub-case u maintains no longer invalid information about v. For the second sub-case we assume that $a.mode = staying$, otherwise we are back to case 1 for the edge (u, a) (and can show that Φ_t decreases). As u forwards the invalid information to a, a in its turn sends $(a, forward)$ to v and deletes v from $a.N$. Therefore, the invalid edge (a, v) vanishes.

4. $v.mode = staying$, $u.(v.mode) = leaving$, $u.mode = staying$:

   If $v \in u.N$, u sends $(u, present)$ to v in $timeout$. Thus, we only need to consider the case in which a message containing the reference of v is $\in u.C$. In this case, u sends $(u, forward)$ to v and the invalid edge vanishes.

Accordingly, in all situations the invalid information is corrected. Therefore, as long as $\Phi_t > 0$ there is a $t' > t$ such that $\Phi_{t'} < \Phi_t$. and eventually all invalid mode information vanishes. □

Lemma 3.4 *During the execution of P' eventually there is no invalid certainty information (i.e., invalid information regarding the certainty variables) in the network.*

Proof. Let u be a node which contains invalid certainty information about a node x, i.e., *u.(x.certainty) =true*, but $u.(x.mode) \neq mode(x)$. There is at least one entry in the $u.mlist$ table in which x appears. Let $(u.mlist).[mh]$ be one of these entries. First we show that eventually all certainty information in $(u.mlist).[mh]$ is set to $true$. That is due to the periodically executed $timeout$ action in which a $present$ message is sent to all nodes $x' \in [x_1, \ldots, x_k]$ stored in $(u.mlist).[mh]$ for which $u.(x'.certainty) = false$. Eventually, x' sends a $(x', forward)$ message and the information becomes $true$. The $m.type$ command which is stored in $(u.mlist).[mh]$ is executed in the $forward$ action and the entry along with x is deleted from $u.mlist$, so the invalid certainty information vanishes. □

Next we show that the network eventually reaches a state, in which all the leaving nodes are gone.

Lemma 3.5 *All leaving nodes eventually execute **exit**.*

Proof. At this point we can assume that all mode information in the network is valid. It remains to show that the proof of Theorem 2.2 is still applicable, i.e., show that eventually $\mathcal{SINGLE}$ evaluates to **true** for a leaving node that has an anchor. Again let x be a node that prevents $\mathcal{SINGLE}$ from doing so, in other words there exists an explicit or implicit edge (x, u). In case the edge is explicit and x is leaving, x behaves similarly to Algorithm 1 and the proof still holds. If the edge is explicit and x is staying, x either directly deletes u (in case $u \notin x.mlist$) or sends $(x, present)$ to u in the $timeout$ action. As shown in Lemma 3.4 these $present$ messages make sure that eventually all certainty information in $x.mlist$ is valid. Once this happens, the entries in which u is stored are deleted from $x.mlist$ and the edge (x, u) vanishes. In case the edge is implicit, we have to distinguish between a staying x and a leaving x. If x is leaving, the $forward$ action is similar to Algorithm 3 and the $present$ action is similar to Algorithm 2, therefore the proof is still applicable. If the message is a $process$ message, u is not saved by x and either a $(x, present)$ or a $(x, forward)$ is sent back to u. So independent of the message type, the edge vanishes eventually. If x is staying, then the $present$ and $process$ actions behave analog to the last sub-case in which x is leaving. The $forward$ message is extended compared to Algorithm 3, but these changes only fuse the implicit edge (x, u) with an explicit edge (x, u), because certainty information is validated. Thus, the implicit (x, u) also vanishes in this case. Therefore, eventually x cannot prevent $\mathcal{SINGLE}$ from evaluating to **true**. Consequently, a leaving node that is not gone yet and has a valid anchor ,is ultimately able to execute **exit**. As long as there are not gone leaving nodes, there exists such a node, which implies that in finite time all nodes eventually execute **exit**.

□

We conclude the proof by showing that P' solves $\mathcal{DP}$ similar to P.

Lemma 3.6 *Once all leaving nodes are gone, P' solves $\mathcal{DP}$.*

Proof. Since there are only staying nodes, the only difference between P' and P is that $m.type$ messages in P are replaced with $process$ messages.

Let u be a node that sends a $(x_1, \dots, x_k, m.type)$ message to v in P. Once v receives the message it executes the corresponding $m.type(x_1, \dots, x_k)$ action. In P', u sends a $(m.type, [x_1, \dots, x_k], process)$ message to v, v saves this information in $v.mlist$ and sends a $(v, present)$ for each x_i. The x_i nodes respond by sending with a $(x_i, forward)$ message to v. Once the last of these $(x_i, forward)$ messages is received by v (or v learns about the valid x_i information by a $(x_i, forward)$ message received as a response to its own self-introduction), the action corresponding to the original $m.type(x_1, \dots, x_k)$ action is executed. Thus, u executes $m.type(x_1, \dots, x_k)$, as in P. □

Lemmas 3.5 and 3.6 prove the statement of Theorem 3.1. □

3.2 FSP for Arbitrary Protocols

Analogous to the results in previous chapter, we can overcome the use of oracles by relaxing the $\mathcal{FDP}$ to the $\mathcal{FSP}$. In order to give a general framework which solves $\mathcal{FSP}$ for arbitrary distributed protocols, we only need to slightly change the protocol framework of Section 3.1. The main differences to the protocol of the previous section is that no oracle is checked in the $timeout$ action and that the **exit** command is not available. Moreover, the behavior of the protocol changes in the $forward$ action as well. We add an additional check that examines whether the reference received in the message is equal to the current anchor. If this is the case, the node falls asleep again by using the **sleep** command and does not perform any further commands. This is necessary, in order for the node to be able to fall permanently asleep, since in that case it will only exchange messages with its anchor.

Since the protocol framework for solving $\mathcal{FSP}$ as well as the correctness proof is very similar to the one for solving $\mathcal{FDP}$ we will omit its detailed description here.

4 Outlook

For future work we definitely see potential in investigating crash failures and message failures. Another interesting direction, is to have stronger requirements than the maintenance of connectivity towards the stabilization process, as for example the guaranty that search and insert node operations execute correctly.

Part II

Efficient topological self-stabilization protocols

In the first part we considered how to handle node departures which happen at some arbitrary time, and presented a general framework, which can be applied to a large class of distributed algorithms, in order for these algorithms to be able to handle node departures, without endangering connectivity. Especially in the case of topological self-stabilizing algorithms, their correctness is not affected at all by the application of this framework. In this part we will see some of these self-stabilizing algorithms, in fact the algorithms presented on the first two chapters fulfill the requirements, that the framework needs in order to be applied.

That fact put aside, in this part our main focus is to develop efficient self-stabilizing protocols for specific network structures. Efficiency does concern the time, as well as message efficiency. Load balancing is also an issue: i.e. that the message overhead is distributed fairly among nodes in the network. The complexity measures we compute hold under the restriction that there is some time point, after which no node departures (as well as joins) happen until the network has stabilized. In this part of the thesis (in contrast to the previous) we restrict ourselves only to the synchronous message passing model, so that we can compute some good upper bounds on the number of steps required for reaching the stable state.

The goal topologies we consider for our self-stabilizing algorithms are the following. In the first chapter of this part we consider an algorithm for self-stabilizingly constructing a variance of a Chord-network, which is a typical network structure used in distributed computing, due to its good routing and structural qualities (low diameter and low degree). The algorithm we give is based on the following work, which was presented in *SPAA (2011)* and later (2014) also published in the Journal of *Theory of Computing Systems, 55(3): 591-612.*:

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

However, the algorithm presented here is an improved version of that work and manages to improve the original work's time complexity from $\mathcal{O}(n \log n)$ to $\mathcal{O}(\log^3 n)$ time steps. Also, the message overhead is improved as well.

In the next chapter we introduce the notion of heterogeneity in distributed systems, since it is reasonable to assume that in many applications the nodes in a network do not always have similar characteristics. We give an example of an algorithm for a heterogeneous distributed storage system. In fact, we focus in this chapter on heterogeneity regarding the node's storage capacity. This chapter is based on the work:

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler: CONE-DHT: A Distributed Self-Stabilizing Algorithm for a Heterogeneous Storage System. DISC 2013

Again, by combining that work with techniques we used also in Chapter 3 (of this part) and adding some new elements, we manage to give an algorithm that stabilizes faster ($\mathcal{O}(\log^3 n)$) in comparison to the one in our original work ($\mathcal{O}(n)$) and moreover maintains a fair load balancing of the message overhead due to routing.

Lastly, we look at one of the simplest network topologies there is, the clique. However, to give an efficient self-stabilizing algorithm with that goal topology is not simple at all. We present an algorithm which has optimal message complexity. This last chapter is based on the work:

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler: A Deterministic Worst-Case Message Complexity Optimal Solution for Resource Discovery. SIROCCO 2013

which appeared at *SIROCCO (2013)*, won the best student paper award, and also appeared later (2015) in the Journal of *Theoretical Computer Science 584: 67-79*.

Chapter 3

Fast-Re-Chord

The first protocol we discuss is called *Fast-Re-Chord*, which is a protocol for self-stabilizingly constructing a $Chord$-like network graph. This chapter is based on the paper

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

Whereas in the original paper the protocol and the analysis were based on completely different techniques, which allowed to show a self-stabilization time of $\mathcal{O}(n \log n)$ time steps, in this chapter we give an improved version of this protocol, which allows to conduct an analysis that results in only $\mathcal{O}(\log^3 n)$ time steps, without increasing the message complexity. So the protocol and the proofs presented in this chapter are completely new and are firstly presented in this thesis.

1 The Chord Network and Its Variants

The Chord system was introduced in an influential paper by Stoica, Morris, Karger, Kaashoek and Balakrishnan [2]. Chord is basically a combination of a hypercubic network with an indexing method called consistent hashing [23], and has the nice property of having logarithmic diameter as well as degree. The Chord overlay network is defined as follows. Let U be the space of all peer addresses and $V \subseteq U$ be the current set of peers (also called *nodes* in the following) with $n = |V|$. There is a (pseudo-)random hash function $h : U \to [0, 1)$ (in Chord, SHA-1) that assigns to each node v an *identifier* $h(v)$ uniformly at random from the $[0, 1)$-interval. The basic structure of Chord is formed by a directed cycle, the so-called *Chord ring*, in which each node connects to its closest successor in the identifier space, where the $[0, 1)$-interval is considered to form a ring. In addition to this, every node v has edges to nodes $p_i(v)$, called *fingers*, with

$$p_i(v) = \text{argmin}\{w \in V \mid h(w) \geq h(v) + 1/2^i (\text{mod}\, 1)\}$$

for every $i \geq 1$. If there is no node $w \in V$ with $h(w) \geq h(v) + 1/2^i (\text{mod}\, 1)$, then the node $w \in V$ with smallest identifier is chosen. In order to route a message from node u to node w, the Chord overlay network uses a path $p(u, v)$ consisting of a sequence of nodes $v_0, v_1, v_2, \ldots, v_\ell$ with the property that $v_0 = u$, for all $j \in \{0, \ldots, \ell - 1\}$, $v_{j+1} = p_{i_j}(v_j)$ where i_j is the smallest integer such that $h(v_{j+1}) \leq h(w)$, and $v_{\ell-1}$ is the first node that has a successor pointer to w. Hence, the path basically represents a binary search strategy and can be shown to be of length at most $\mathcal{O}(\log n)$ with high probability (given that the nodes have random identifiers).

Several variants of Chord have already been studied since the presentation of the Chord network. In [88] a variant called EPI Chord is presented that allows the system to do parallel searches for the best route to the search key. This does not improve the asymptotically worst-case cost of $\mathcal{O}(\log n)$ messages of Chord but can achieve $\mathcal{O}(1)$ hop lookup performance under lookup-intensive workloads due to caching. In [89] 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 [90] and [91], where links to the predecessors are stored instead of only links to the successors of a node. In [91] also the physical distance is taken into account to estimate the shortest route. All these variants only care about the lookup cost, but present no self-stabilizing process to maintain the Chord structure. In [92] 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.

1.1 Fast-Re-Chord

Here we present *Fast-Re-Chord*, a self-stabilizing variant of Chord. In the Fast-Re-Chord network, each node u representing a peer has an identifier $u.id \in [0,1)$, that defines its position in the $[0,1)$-interval. The self-stabilization mechanism is purely local in that a node only has to inspect its local state in order for the algorithm to work. No global knowledge of the network is needed. Our main result is the following.

Theorem 1.1 *Our proposed protocol stabilizes after $\mathcal{O}(\log^3 n)$ rounds from any weakly connected state w.h.p. The stable state of the protocol contains Chord as a sub-graph, so it can faithfully emulate any applications on top of Chord.*

Moreover, isolated join and leave requests can be handled in $\mathcal{O}(log^2 n)$ time steps with the self-stabilization mechanism of Fast-Re-Chord.

More particularly, our goal structure is the *Fast-Re-Chord* graph, i.e. $G^{\textit{Fast-Re-Chord}}$, which is a variant of Chord, where each node additionally contains a few shortcuts to the nearest nodes in the network. This is done in order to speed up the self-stabilization time, whereas it does not asymptotically increase the size of the neighborhood.

More specifically, each node u in $G^{\textit{Fast-Re-Chord}}$ maintains a neighborhood consisting of the $fingers$ and the $direct$ neighbors.

The $fingers$ which u maintains are defined as in the original $Chord$ protocol, with the addition that they are maintained at both sides (left and right fingers). So the fingers of a node u in $G^{\textit{Fast-Re-Chord}}$ consist of the nodes

$$fright_i(u) = \mathrm{argmin}_{\mathrm{w}\in\mathrm{V}}\{w.id \geq u.id + 1/2^i (\mathrm{mod}\, 1)\}$$

as well as the nodes

$$fleft_i(u) = \mathrm{argmax}_{\mathrm{w}\in\mathrm{V}}\{w.id \leq u.id - 1/2^i (\mathrm{mod}\, 1)\}$$

for every $i \geq 1$.
Also let $Fright(u) = \bigcup_{i\geq 0} fright_i(u)$ and $Fleft(u) = \bigcup_{i\geq 0} fleft_i(u)$.

The $direct$ neighbors also exist on both sides (left and right direct neighbors), and they are defined as the sets $rDirect(u)$ and $lDirect(u)$ of the closest $6u.mylogn$ neighbors to the right (resp. to the left), where $u.mylogn$ is a variable that denotes u's estimation of $\log(n)$, and is computed as $u.mylogn = \max\{k \in \mathbb{N} : \exists w \in \text{neighborhood of } u : |w.id - u.id| \leq 1/2^k\}$.

So $G^{\textit{Fast-Re-Chord}}$ is defined as

$$G^{\textit{Fast-Re-Chord}} = (V, E = E_e \cup E_i) : E_e = (u, v) : v \in Fright(u) \cup Fleft(u) \cup rDirect(u) \cup lDirect(u)$$

Moreover, for the sake of the analysis, we will need the definition of the sorted-list graph, which is a graph, where each node stores as a neighbor its closest node from the left as well as from the right.

$$G^{List} = (V, E_e \cup E_i) : E_e = (u, v) : v = \text{argmin}_{w \in V}\{w.id \geq u.id\} \vee v = \text{argmax}_{w \in V}\{w.id \leq u.id\}$$

Chord has two kinds of edges, successor-predecessor edges that form the Chord-ring, as well as fingers. In the stable state each node in Fast-Re-Chord has an edge to its closest right and closest left neighbor, which would be the successor and predecessor of that node in Chord. In Chord, each node u has a finger edge that connects u with the node having a value closest to (and greater than) $u + \frac{1}{2^i} (\mod 1)$, in a clockwise direction along a $[0, 1)$ circle, for different values of i. Fast-Re-Chord achieves the same. Therefore, each edge of Chord is included in Fast-Re-Chord, which implies the following.

Corollary 1.2 *In the stable state, Chord is a sub-graph of Fast-Re-Chord.*

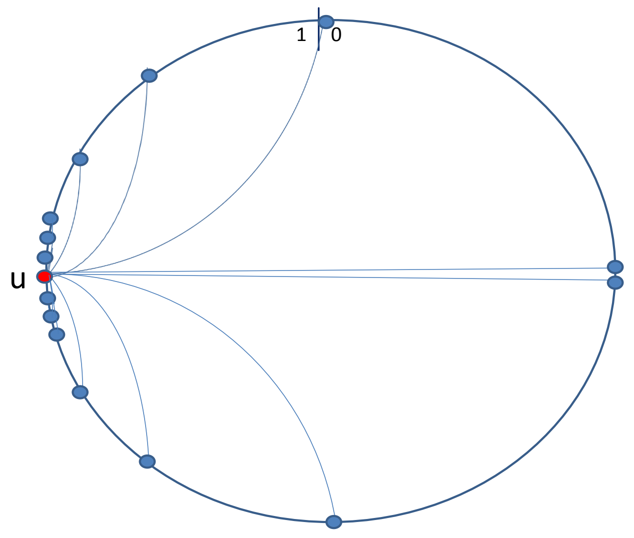


Figure 3.1: In this picture we depict the $[0, 1)$ interval as a closed circle. A node u maintains in the *Fast-Re-Chord* network edges to its fingers, both on the left and on the right, as well as to its closest $u.mylogn$ nodes, both on the left and on the right of u.

2 Algorithm

The goal of the algorithm is to reach the desired Chord-structure in a self-stabilizing manner.

Our approach is based on a procedure called *linearization*, which has been used to form a sorted list in various papers [35][34]. The basic idea of this procedure is that each node u forwards the id of each node v it is aware of, but which is not needed for u's neighborhood to some node in u's neighborhood which is closer to v. In addition to this classic linearization technique, we add an extra rule. If u gains a new neighbor v and stores v in its neighborhood, it also introduces its whole neighborhood to v. In that way the forming of the correct neighborhood of v is being sped up, without increasing the message complexity in the stable state. That additional rule is crucial for the analysis which results in a polylogarithmic number of convergence steps.

The algorithm consists of a series of periodically executed actions, as seen in algorithm 29. Among these actions is the reading of the channel $u.C$ for incoming messages. If there are any, then the corresponding methods are executed: for example, if there is an incoming message of the form $(m.id, lin)$, the method $linearize(m.id)$ is executed.

A technique we use in the algorithm is the probing. Periodically, each node u sends a probing message (for each direction, left and right) which goes through all of its neighbors. The purpose of this procedure is to test if u is connected through list-edges to them. Each node u uses a left and a right probing message. If there is currently no right (resp. left) probing procedure going on, one is started, as we can see in the actions of the algorithm. The probing procedure continues in each round as each node u tries to contact the node being closest to the current probing goal, through sending a $requestprobingright$ and $requestprobingleft$ message respectively through the $probingr$ (resp. $probingl$) actions. A node that receives a $requestprobingright$ (or $requestprobingleft$) message responds to u by sending the node that is closer to the current probing goal through a $probingr$ (or $probingl$) message and the same procedure occurs again until the last probing goal is reached. Note that a probing procedure is induced every $\Theta(|\text{fingers of } u|)$ steps, which results in having probing messages more often when the neighborhood is small, but also not creating too many probing messages when the neighborhood is large, for example at the stable state, thus keeping the message overhead low.

As we require a ring as a sub-structure, we need a further operation to form it. The basic idea of this procedure is that each node maintains the variables $u.rring$ and $u.lring$ to store a ring edge, that is an edge that connects the minimal node with the maximal node and vice versa. A node u sends periodically $requestleftring$ (resp. $requestrightring$) to the node its ring edge is connected to, which responds such that u can update it by sending a smaller (resp. larger) node.

Moreover, during the periodic actions, u also runs the $linearize()$ method, which updates u's neighborhood, keeps the nodes needed as fingers or direct neighbors, and linearizes the nodes that are not needed by using the $forward$ method.

At last, a $validitycheck$ is made, where it is checked whether the variables maintained by u really fulfill the restrictions they ought to (for example, the edge to the right ring node of u must be to the right of u).

2.1 Variables of the Protocol

Here we present the variables each node maintains in our protocol. The fingers of a node u are stored in the variables $u.rfinger_i$, with $u.firstf \leq i \leq u.mylogn$, where $u.firstf$ is the order of the closest finger to u. More specifically, the order of the closest finger is computed as

$u.firstf = \min\{k \in \mathbb{N} : \nexists w \in \Gamma(u) : |w.id - u.id| \leq 1/2^{u.mylogn-k+1}\}$. We say that $u.rfinger_i$ (resp. $u.lfinger_i$) is the right (resp. left) finger of u of order i, and this is set as (as we can see in the pseudocode) $u.rfinger_i = \text{argmin}_{w \in \Gamma(u)}\{w.id : w.id \geq u.id + 1/2^{u.mylogn-i+1} (\bmod 1)\}$ (resp. $u.lfinger_i = \text{argmax}_{w \in \Gamma(u)}\{w.id : w.id \leq u.id - 1/2^{u.mylogn-i+1} (\bmod 1)\}$).

The list of all variables a node u maintains is

- $u.mylogn$: The estimation of $\log(n)$.
- $u.firstf$: The order of the first finger, computed respectively to the "closest" node to u from u's neighborhood
- $u.rfinger_i, \forall i$ with $u.firstf \leq i \leq u.mylogn$: The right finger of node u of order i.
- $u.lfinger_i, \forall i$ with $u.firstf \leq i \leq u.mylogn$: The left finger of node u of order i.
- $u.rDirect$: The set of nodes in u's neighborhood, being "closest" to u from the right.
- $u.lDirect$: The set of nodes in u's neighborhood, being "closest" to u from the left.
- $u.rring$: The right ring edge of u.
- $u.lring$: The left ring edge of u.
- $u.P$: The nodes used for supporting the probing process of u.
- $u.nextr$ A pointer that points to the next node of the right probing process.
- $u.nextl$ A pointer that points to the next node of the left probing process.
- $u.rprobcounter$ Counts the number of steps the right probing process currently lasts
- $u.lprobcounter$ Counts the number of steps the left probing process currently lasts

$u.Rfingers$ (resp. $u.Lfingers$) is also used to denote the set of all right fingers (resp. left fingers). Moreover, $u.N$ denotes the neighborhood of u, that includes all the nodes stored in u, i.e. the nodes in $u.Rfingers$,$u.Lfingers$, $u.rDirect$, $u.lDirect$, $u.P$, as well as $u.rring$ and $u.lring$. Note also that sometimes instead of a variable $u.var$ in the pseudo-code we also write just var.

2.2 The Protocol

In this sub-section the pseudocode of the protocol is presented.

Algorithm 8 Periodic Actions of node u

```
true→
u.lprobcounter:=u.lprobcounter+1
u.rprobcounter:=u.lprobcounter+1
if $u.rprobcounter > 82u.mylogn$ then                    ▷ If the probing ended, start probing
    u.probingr($\min\{w \in u.N : w.id > u.id\}, u$)
    u.rprobcounter=0
if $u.lprobcounter > 82u.mylogn$ then
    u.probingl($\max\{w \in u.N : w.id < u.id\}, u$)
    u.lprobcounter=0
if $u.id \geq w.id, \forall w \in u.N$ then
    send message(u,requestleftring) to u.lring
else
    linearize(u.lring)
    u.lring=null
if $u.id \leq w.id, \forall w \in u.N$ then
    send message(u,requestrightring) to u.rring
else
    linearize(u.rring)
    u.rring=null
message $m \in u.C$ →
if m.type=lin then
    linearize(m.id)
else if m.type=probingr then
    probingr(m.id,m.sender)
else if m.type=probingl then
    probingl(m.id,m.sender)
else if m.type=requestprobingright then
    continueprobingright(m.sender,stage)
else if m.type=requestprobingleft then
    continueprobingleft(m.sender,stage)
else if m.type=requestrightring then
    rightring(m.id)
else if m.type=requestleftring then
    leftring(m.id)
else if m.type=requestneib then
    requestneib(m.id)
linearize()
validitycheck() ▷ Check whether the nodes in $u.N$ are within the boundaries they are supposed to,
else linearize them (is a trivial procedure and omitted here)
```

Algorithm 9 U.LINEARIZE(V)

```
if $v \notin u.N \vee v = \emptyset$ then
    $u.Nold = u.N$
    $u.N = u.N \cup \{v\}$

    if $u.id \leq w.id, \forall w \in u.N$ then                ▷ Update ring edges
        u.rring=max$\{w \in u.N\}$
    else
        if $u.rring \neq null$ then
            u.linearize(u.rring)
        u.rring=$null$
    if $u.id \geq w.id, \forall w \in u.N$ then
        u.lring=max $w \in u.N$
    else
        if $u.lring \neq null$ then
            u.linearize(u.lring)
        u.lring=$null$

    $u.mylogn = \max\{k \in \mathbb{N} : \exists w \in u.N : |w.id - u.id| \leq 1/2^k\}$    ▷ estimate log(n)
    $u.firstf = \min\{k \in \mathbb{Z} : \nexists w \in u.N : |w.id - u.id| \leq 1/2^{mylogn-k+1}\}$
    for all $i : u.firstf \leq i \leq mylogn$ do                ▷ compute right and left fingers
        $u.rfinger_i = \text{argmin}_{w \in u.N}\{w.id : w.id \geq v.id + 1/2^{mylogn-i+1} (\text{mod}\, 1)\}$
        if $u.rfinger_i = null$ then
            $u.rfinger_i = \text{argmin}_{w \in u.N}\{w.id\}$
        $u.lfinger_i = \text{argmax}_{w \in u.N}\{w.id : w.id \leq v.id - 1/2^{mylogn-i+1} (\text{mod}\, 1)\}$
        if $u.lfinger_i = null$ then
            $u.lfinger_i = \text{argmax}_{w \in u.N}\{w.id\}$
    u.Rfingers=$\bigcup_{\forall firstf \leq i \leq mylogn} u.rfinger_i$
    u.Lfingers=$\bigcup_{\forall firstf \leq i \leq mylogn} u.lfinger_i$

                        ▷ 6mylogn nodes closest to $u$ from the right and the left
    $u.rDirect = \{w \in u.N : |\{v \in u.N : x \leq y$, where $x = \min\{v.id - u.id, 1 - u.id + v.id\}, y = \min\{w.id - u.id, 1 - u.id + w.id\} \wedge x, y > 0\}| \leq 6mylogn\}$
    $u.rDirect = \{w \in u.N : |\{v \in u.N : x \leq y$, where $x = \min\{u.id - v.id, 1 - v.id + u.id\}, y = \min\{u.id - w.id, 1 - w.id + u.id\} \wedge x, y > 0\}| \leq 6mylogn\}$

    for all $w \in u.N : u \notin Rfingers \cup Lfingers \cup rDirect \cup lDirect \cup \{rring\} \cup \{lring\} \cup u.P$ do
        u.forward(w)                ▷ get rid of the rest

    $u.N = Rfingers \cup Lfingers \cup rDirect \cup lDirect \cup \{rring\} \cup \{lring\} \cup u.P$
    if $u.N \neq u.Nold$ then                ▷ If v was included in new u.N
        for all $w \in u.N$ do
            send message (w,lin) to $v$
        send message (v, requestneib) to h=$\{u.Nold.finger_j : v = u.finger_j\}$
    else
        u.forward(v)
```

Algorithm 10 U.PROBINGR(V,M.SENDER)

```
if $v.id > (m.sender).id$ then
    for all $w \in u.P : (m.sender).id < w.id < v.id$ do                ▷ Update u.P
        $u.P := u.P/\{w\}$
        u.linearize(w)
else
    for all $w \in u.P : ((m.sender).id < w.id) \vee (w.id < v.id)$ do
        $u.P := u.P/\{w\}$
        u.linearize(w)
if $v \notin u.N$ then
    $u.P = u.P \cup \{v\}$
    u.linearize(v)
if $m.sender \notin u.N$ then
    $u.P = u.P \cup \{m.sender\}$
    u.linearize(m.sender)

first=min$\{w \in u.N : w.id > u.id\}$
if first=null then
    first=u.lring
u.nextr=min$\{w \in Rfingers \cup rDirect : w.id > first.id\}$
if next=null then
    u.nextr=min$\{w \in Rfingers \cup rDirect\}$
if $u.nextr \in rDirect$ then
    stage=direct
else
    stage=fingers
if $v = u.rfinger_{mylogn}$ then                ▷ probing ended, restart probing
    u.rprobcounter=0
    send message (requestprobingright,stage) to $first$
else if $u.rprobcounter > 82u.mylogn$ then      ▷ probing counter reached limit, restart probing
    for all $w \in Rfingers \cup rDirect$ do
        send message (u,lin) to $w$
    u.rprobcounter=0
    send message (requestprobingright,stage) to $first$
else                                            ▷ continue probing with searching for next node
    u.nextr=min$\{w \in Rfingers \cup rDirect : w.id > v.id\}$
    if next=null then
        u.nextr=min$\{w \in Rfingers \cup rDirect\}$
    if $u.nextr \in rDirect$ then
        stage=direct
    else
        stage=fingers
    send message (requestprobingright,stage) to $v$
```

Algorithm 11 U.PROBINGL(V,M.SENDER)

```
if $v.id < (m.sender).id$ then
    for all $w \in u.P : (m.sender).id > w.id > v.id$ do        ▷ Update u.P
        $u.P := u.P/\{w\}$
        u.linearize(w)
else
    for all $w \in u.P : ((m.sender).id > w.id) \vee (w.id > v.id)$ do        ▷ Update u.P
        $u.P := u.P/\{w\}$
        u.linearize(w)
if $v \notin u.N$ then
    $u.P = u.P \cup \{v\}$
    u.linearize(v)
if $m.sender \notin u.N$ then
    $u.P = u.P \cup \{m.sender\}$
    u.linearize(m.sender)

first=max$\{w \in u.N : w.id < u.id\}$
if first=null then
    first=u.lring
u.nextl=max$\{w \in Lfingers \cup lDirect : w.id < first.id\}$
if next=null then
    u.nextl=max$\{w \in Lfingers \cup lDirect\}$
if $u.nextr \in lDirect$ then
    stage=direct
else
    stage=fingers
if $v = u.lfinger_{mylogn}$ then        ▷ probing ended, restart probing
    u.lprobcounter=0
    send message (requestprobingleft,stage) to $first$
else if $u.lprobcounter > 82u.mylogn$ then        ▷ probing counter reached limit, restart probing
    for all $w \in Lfingers \cup lDirect$ do
        send message (u,lin) to $w$
    u.rprobcounter=0
    send message (requestprobingleft,stage) to $first$
else        ▷ continue probing with searching for next node
    u.nextl=max$\{w \in Lfingers \cup lDirect : w.id < v.id\}$
    if next=null then
        u.nextl=max$\{w \in Lfingers \cup lDirect\}$
    if $u.nextr \in lDirect$ then
        stage=direct
    else
        stage=fingers
    send message (requestprobingleft,stage) to $v$
```

Algorithm 12 U.FORWARD(V)

```
                                        ▷ forward v to the closest neighbor
if $v.id > u.id$ then
    if $v.id > z = \min\{w \in u.N : w.id > u.id\}$ then
        send message (v,lin) to x=max$\{w \in u.N : w.id < v.id\}$
        send message (min$\{w \in u.N : w.id > u.id\}$,lin) to $v$
    else
        send message (u,lin) to $v$
else if $v.id < u.id$ then
    if $v.id < z = \max\{w \in u.N : w.id < u.id\}$ then
        send message (v,lin) to x=min$\{w \in u.N : w.id > v.id\}$
        send message (max$\{w \in u.N : w.id < u.id\}$,lin) to $v$
    else
        send message (u,lin) to $v$
```

Algorithm 13 U.CONTINUEPROBINGRIGHT(M.SENDER,STAGE)

```
        ▷ Sends the next probing node to the sender, so that the probing can be continued
if stage=fingers then
    goal=min$\{(m.sender).id + 1/2^i (\bmod 1) | (m.sender).id + 1/2^i (\bmod 1) > u.id\}$
    send message(argmin$_{w \in u.N/u.P}\{|w.id < goal|\}$,probingr) to m.sender
else
    send message(argmin$_{\forall y \in u.rDirect}|y.id - u.id|$,probingr) to m.sender
```

Algorithm 14 U.CONTINUEPROBINGLEFT(M.SENDER,STAGE)

```
        ▷ Sends the next probing node to the sender, so that the probing can be continued
if stage=fingers then
    goal=max$\{(m.sender).id - 1/2^i (\bmod 1) | (m.sender).id - 1/2^i (\bmod 1) < u.id\}$
    send message(argmin$_{w \in u.N/u.P}\{|w.id < goal|\}$,v,probingl) to m.sender
else
    send message(argmin$_{\forall y \in u.lDirect}|y.id - u.id|$,probingr) to m.sender
```

Algorithm 15 U.RIGHTRING(V)

```
send message (max$\{w \in u.N\}$,lin) to $v$            ▷ respond to a rightring request
```

Algorithm 16 U.LEFTRING(V)

```
send message (min$\{w \in u.N\}$,lin) to $v$             ▷ respond to a leftring request
```

Algorithm 17 U.REQUESTNEIB(V)

```
for all $w \in u.N$ do
    send message (w,lin) to $v$                       ▷ respond to a requestneib
```

3 Analysis

In order to show that our protocol is a self-stabilizing algorithm for $G^{Fast-Re-Chord}$, we need to show two main properties. The convergence, which means that the $G^{Fast-Re-Chord}$ is reached out of any weakly-connected state, and the closure, which means that once a $G^{Fast-Re-Chord}$ state is reached, it also stays that way. We show that the convergence can be achieved in a polylogarithmic number of time steps. Furthermore, we show the message complexity for both the time of the convergence, as well as during the closure.

Definition 3.1 *First we give the following definitions.*

- *The $distance$ between two nodes u,v ($distance(u, v)$) with $u.id > v.id$ is defined as* $\min\{u.id - v.id, 1 - u.id + v.id\}$.
- *We denote as $p_t(u, v)$ an $undirected\ path$ that exists between node u,v at some time step t.*
- *Let a, b, c be three consecutive nodes of some $undirected\ path$ $p_t(u, v)$. Then b is called a (left) $corner\ node$ of $p_t(u, v)$ if $a.id > b.id$ and $c.id > b.id$. It is called a (right) $corner\ node$ of $p_t(u, v)$ if $a.id < b.id$ and $c.id < b.id$.*
- *A $linear\ path$ between two nodes u,v is an $undirected\ path$ without any $corner\ nodes$ (despite possibly the first and last node of the path).*
- *The right (resp. left) $link$ of u of order i ($rlink_i(u)$ (resp. $llink_i(u)$)) is the smallest (resp. largest) node in u's neighborhood ($u.N$) having larger (resp. smaller) id than $u.id + 1/2^{\lceil log(n) \rceil - i+1}$ (mod 1) (resp. smaller than $u.id - 1/2^{\lceil log(n) \rceil - i+1}$ (mod 1)).*

  If there is not such a node then $rlink_i(u)$ (resp. $llink_i(u)$) is equal to $rlink_{i-1}(u)$ (resp. $llink_{i-1}(u)$), and if there does not even exist a node from u to $u.id + 1/2^{\lceil log(n) \rceil - i+1}$ (mod 1) (resp. $u.id - 1/2^{\lceil log(n) \rceil - i+1}$ (mod 1)), then $rlink_i(u)$ (resp. $llink_i(u)$) is equal to u. Note that the value of $rlink_i(u)$ (resp. $llink_i(u)$) depends on the current values in $u.N$.
- *The correct right (resp. left) $finger$ of u of order i ($rfinger_i(u)$ (resp. $lfinger_i(u)$)) is equal to the value $u.rfinger_i$ (resp. $u.lfinger_i$) would get through the $linearize()$ action if $u.N = V$. In other words, the value these variables must have at the stable state.*

3.1 Preliminary Properties

Next we show some structural properties of the network.

Lemma 3.2 *The number of nodes in an interval of $1/n$ is at most $6 \log n$, w.h.p..*

Proof. We consider the process of distributing n nodes in the $[0, 1)$ interval (since we assume that the ids are distributed uniformly at random), as throwing n balls into n bins of size $1/n$ each. Then we can use the well-known fact that a bin has at most $\frac{8\ln(n)}{\ln(\ln(n))}$ number of balls w.h.p., from which it follows that the number of nodes are no more than $\frac{8\ln(n)}{\ln(\ln(n))} \leq 8\ln(n) = \frac{8\log(n)}{\log(e)} \leq 6\log n$ w.h.p.. □

Lemma 3.3 *$u.mylogn \leq 3\log n$ w.h.p. and if u knows at least 1 node within $1/n$ distance from itself, then $u.mylogn > \log n$. Moreover, the number of u's fingers/links is at most $14\log n$ w.h.p..*

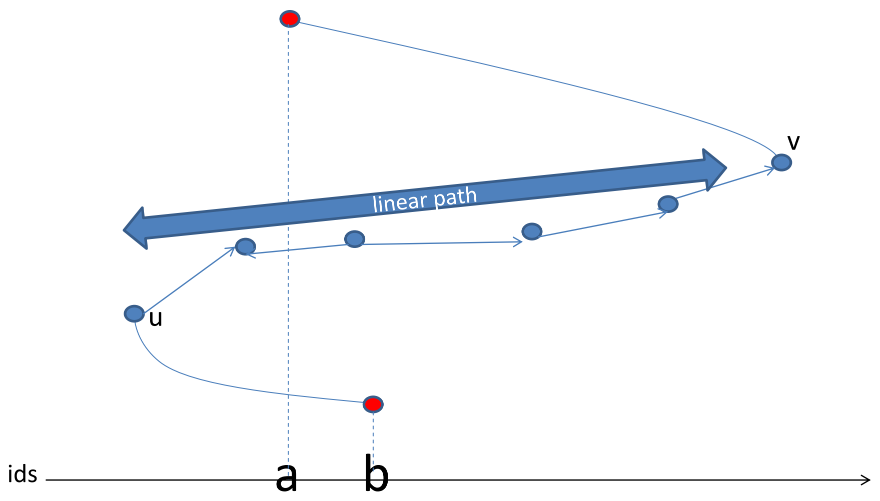


Figure 3.2: We assume that the nodes on the picture are ordered from left to right according to their ids. let's consider two consecutive nodes a and b and a possible path between them. The corner nodes in that path are u and v and for example the undirected path $p(u,v)$ is a linear path, since it does not contain any corner nodes besides the first and last node.

Proof. According to the protocol $u.mylogn$ is calculated as $\max\{k \in \mathbb{N} : \exists w \in u.N : |w.id - u.id| \leq 1/2^k\}$. That means in order for $u.mylogn$ to be larger than $3\log n$ there must exist a w in the network which is within the interval $I = [u.id - 1/2^{3\log n}, u.id + 1/2^{3\log n}]$. The probability that this does not happen is $Pr[\nexists w \in I] \geq 1 - n\frac{2}{2^{3\log n}} = 1 - \frac{2}{n^2} \rightarrow 1$. The second statement of the lemma follows directly from the computation of $u.mylogn$ in the $linearize$ action of the algorithm. Now we consider the third statement. Due to Lemma 3.2 the number of nodes within a range of $1/n$ away from $u.id$ are at most $12\log n$ w.h.p. ($6\log n$ at the right and $6\log n$ at the left), which means that the number of fingers of order less than 1 is at most $12\log n$ w.h.p., so all in all the number of fingers/links is at most $14\log n$ w.h.p. ($7\log n$ at each side). □

Before presenting the next lemma, note that by a successful probing we mean that all nodes in $u.N$ are reached within a probing procedure, without the $u.lprobcounter$ or $u.rprobcounter$ to reach its maximum value in the $u.probingl$ or $u.probingr$ action.

Lemma 3.4 *In the stable state, the size of the neighborhood of a node u ($u.N$) is $\mathcal{O}(\log n)$ w.h.p. and the probing succeeds w.h.p.*

Proof. Let u be a node in a graph being in the stable state. That means that u maintains its correct neighborhood, i.e. $u.N = N(u)$ (for example $u.rfinger_1 = rfinger_1(u)$).

Let X_i be the random variable that is equal to the number of nodes in $u.P$ between y, the (w.l.o.g.) right link of u of order i and w, the one of order $i+1$. In order to get from y to w through the probing process, the closest node to w is continuously sought. More specifically, y delegates the probing message to its right link of order i, z (if not it means it has a neighbor even closer to w which makes the probing even faster). In case $rlink_{i+1}(u) = z$, then the probing from link i to $i+1$ just takes 2 rounds. If this is not the case, because $z.id > w.id$, then we have to count the number of nodes between z and w.

Now we compute the expected distance between z and w, which is at most the distance between z (or $z.id$) and $u.id + 1/2^{log(n)-(i+1)+1}$ (mod 1). Note that y is the leftmost node after $u.id + 1/2^{log(n)-i+1}$(mod 1) and z the leftmost node after $y.id + 1/2^{log(n)-i+1}$(mod 1). That means that $E[z.id] = E[z.id - y.id + y.id] = E[z.id - y.id] + E[y.id]= 1/2^{log(n)-i+1}+1/n + u.id + 1/2^{log(n)-i+1} + 1/n$ (mod 1) $= u.id + 1/2^{log(n)-(i+1)+1} + 2/n$ (mod 1). So the mean length of the interval $[z.id, u.id + 1/2^{log(n)-(i+1)+1}$ (mod 1)$]$ is $2/n$, in other words, the interval between z and w has expected size at most $2/n$. Due to Lemma 3.2, the number of nodes in this interval is at most $12 \log n$ w.h.p., and since each node f in this interval has at least one neighbor at distance $\leq 1/n$ (either that, or there are very few nodes in the interval w.h.p.), due to Lemma 3.3 it holds that $f.mylogn > \log n$, and the size of $f.rDirect$ and $f.lDirect$ is at least $6 \log n$. That means that the probing procedure from z to w takes at most 2 hops at expectation (since at each hop at least $6 \log n$ nodes are left behind). So all in all, the probing from $y = u.rlink_i$ to $w = u.rlink_{i+1}$ takes at most 3 hops at expectation, i.e. $E[X_i] \leq 3$.

Let X be the random variable that denotes the number of probing hops due to the fingers of u. Summing up over all (w.l.o.g.) right links (which we know are at most $7 \log n$ w.h.p. due to Lemma 3.3), we have that the expected number of probing hops is $E[X] \leq E[\sum_{\forall i} X_i] = \sum_{\forall i} E[X_i] = 3 \cdot 7 \log n = 21 \log n$.

By using Hoeffding's inequality we have that $P(X - E[X] \geq \log n) \leq e^{-\frac{2n^2 \log^2 n}{\sum_{j=1}^{n}(2\log n - 1)^2}} \leq e^{-\frac{2n^2 \log^2 n}{\sum_{j=1}^{n}(2\log n)^2}} = e^{-\frac{2n^2 \log^2 n}{4n \log^2 n}} = e^{-\frac{n}{2}} \to 0$. So the number of hops due to the fingers is no more than $22 \log n$ w.h.p., which means that if we take into account that the number of hops is equal to the number of hops due to the fingers plus the hops due to the nodes in $u.rDirect$ (or $u.lDirect$) the total number of probing hops is w.h.p. at most $22 \log n + 6u.mylogn \leq 22 \log n + 6(3 \log n)$ (due to Lemma 3.3) $= 40 \log n$. That means that $|u.P| = O(\log n)$ w.h.p., from which we can derive that $|u.N| = O(\log n)$ w.h.p., as due to Lemma 3.3 we also know that u's fingers are at most $14 \log n$ w.h.p..

The number of timesteps required for probing is $80 \log n$ (double the number of hops) w.h.p., since for each hop there is a probing message sent as well as a response ($continueprobing$) message. In our case, according to Lemma 3.3, $80 \log n < 80u.mylogn$, That means that probing succeds w.h.p., according to the pseudocode of the probing procedure (Algorithms 10, 11).

□

3.2 Convergence

The main theorem we need to show is the following.

Theorem 3.5 *After O($log^3 n$) steps, the network has reached a state, where the explicit network graph is a supergraph of $G^{\text{Fast-Re-Chord}}$.*

First of all, note that, out of any initial weakly-connected configuration, at most after one round, after the $validitycheck()$ has been called by each node in the network, all nodes maintain valid neighborhoods. We consider this timestep, in which each node maintains a valid neighborhood as t_0.

The analysis of the convergence continues in several phases. Our first goal is to show that the sorted list is formed as a substructure in the network. We do that by showing that the path length between two contiguous nodes is halfed every $O(\log^2 n)$ timesteps. More precisely we show that each linear subpath of that path is halfed. In order to do that we show that each node on that subpath maintains its links up to a specific order on that subpath, and this order bound increases every constant number of rounds.

Next we need to show how the Fast-Re-Chord graph is formed out of the sorted list. It is easy to show inductively that on the sorted list the correct fingers of a node u are formed, one after the other. At last, we show that explicit edges that are not part of the Fast-Re-Chord graph disappear, and as a result the Fast-Re-Chord graph is being formed.

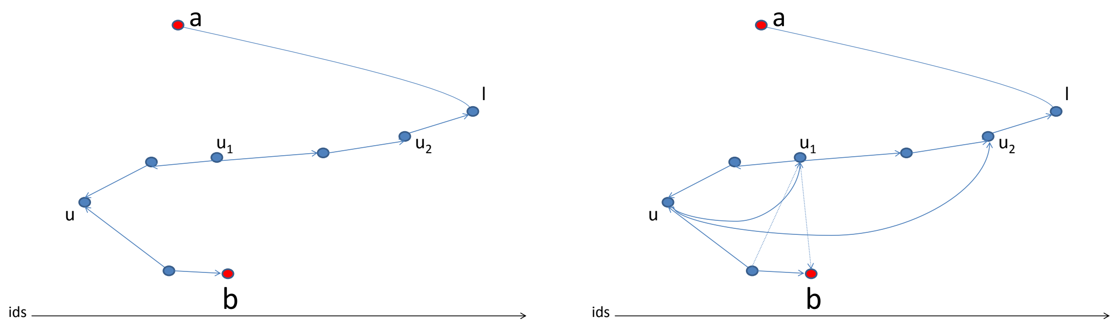


Figure 3.3: let's consider the situation on the left. After $O(\log n)$ rounds (Lemma 3.6) node u will have its links u_1 and u_2 at the path $p(u, l)$ as well as at the path $p(u, b)$. This means that u_1 will be also on $p(u, b)$ (right picture) and the subpath from l to b shrinks.

Phase 1: Reaching the Sorted List

We first show the following lemma.

Lemma 3.6 *Let a timestep* $t \geq t_1 = 82 \log n + t_0$ *and a node* u *being on a linear path* $\in lp_t(a, b)$. *Then at time* $t_j = t_1 + i \cdot h, i > 0$ *(for a sufficiently large constant* h*) it holds w.h.p. that* u*, as well as its right (resp. left) links (if it has any of them) of order* $\leq i$ *such that* $1/2^{\lceil log(n) \rceil - i + 1}$ *(i.e. the length of the i-th order right, resp. left link) is smaller than* $\frac{b.id - u.id}{2}$ *(resp.* $\frac{u.id - a.id}{2}$*), are on a linear path between* a *and* b.

Proof.

The proof is done through induction over i.

Induction basis :

We start the induction at round $t_1 = 82\log n + t_0$. As we have shown, after $82\log n$ rounds all the nodes will have already had a complete probing procedure and will have reset their probing procedure, and at least one new has been started. W.l.o.g. we restrict our analysis to the case of the right links. If u already has a right link of order 1 at a linear path $p \in lp_{t_1}(a,b)$ we are done. So let us assume this is not the case.

Let v be the node of p, being the closest one to u from the right. In case u does not have an edge to v (then obviously v has an edge to u) we consider following subcases. If (v,u) is an explicit edge, then v will contact u during its next probing procedure, which happens in $O(\log n)$ rounds w.h.p. If (v,u) is an implicit edge, then following cases are possible.

- The edge can be a (u, lin) message, in which case v sends u a lin message with v's closest neighbor from the left.
- This edge can be a $(u, probingr)$ message, in which case v would linearize u or contact u during the next probing otherwise if u is the first argument. If u is the $m.sender$ argument, it is handled analogously.
- The implicit edge (v,u) could be a $requestprobing$ message then a $continueprobing$ message is sent back to u.
- The edge can otherwise be a $(u, rightring)$ message, in which case it means that u sent that message and has a ring edge to v. In case the ring edge was dropped it would have been linearized before by u.
- The edge could be an $(u, requestneib)$ message, in which case v sends $v.N$ to u.

.

Now, either is v the first direct right neighbor of u (the leftmost of $u.rDirect$) or u maintains another node which is that, let that be w. Either during the last probing procedure of u, or when u learns about v, v is introduced to w, which means that after a constant number of rounds the first direct right neighbor of u is at a linear path from a to b. By continuing this argumentation analogously, we conclude that all nodes in $u.rDirect$, as well as all nodes in $u.P$ are between u and $rlink_1(u)$ (including $rlink_1(u)$) are at a linear path from a to b after $O(\log n)$ rounds w.h.p. (since there lie $O(\log n)$ nodes between u and $rlink_1(u)$ w.h.p.).

Induction step:

According to the induction hypothesis, at time $t_1 + (i-1)h$, u has already links of order $< i$ at some $p_x \in lp_{t_1+(i-1)h}(a,b)$.

If the link $rlink_i(u)$ is already on the path or $1/2^{\lceil log(n)\rceil - i+1} > \frac{b.id-u.id}{2}$, then we are done. Else, we know from the induction hypothesis that u maintains $rlink_{i-1}(u)$ at some $p_y \in lp_{t_1+i-1}(a,b)$, if u was already on such a path. Also, according to the induction hypothesis, $rlink_{i-1}(u)$ maintains also its link $rlink_{i-1}(rlink_{i-1}(u))$ on some path $\in lp_{t_1+i-1}(a,b)$ (see Figure 3.2), if $1/2^{\lceil log(n)\rceil - i} < \frac{b.id-rlink_{i-1}(u)}{2}$. (If this is not the case then also $1/2^{\lceil log(n)\rceil - i+1} > \frac{b.id-u.id}{2}$, and then we are done.) Now, one case is if $rlink_{i-1}(rlink_{i-1}(u))$ and u have been introduced to each other by $rlink_{i-1}(u)$, at timestep $t_j > t_0$ when $rlink_{i-1}(u)$ learned about u (or $rlink_{i-1}(rlink_{i-1}(u))$), according to the $(rlink_{i-1}(u)).linearize(u)$ action (or $(rlink_{i-1}(u)).linearize(rlink_{i-1}(rlink_{i-1}(u)))$ action). Then the two nodes are introduced to each other because $(rlink_{i-1}(u)).N$ would be unequal to

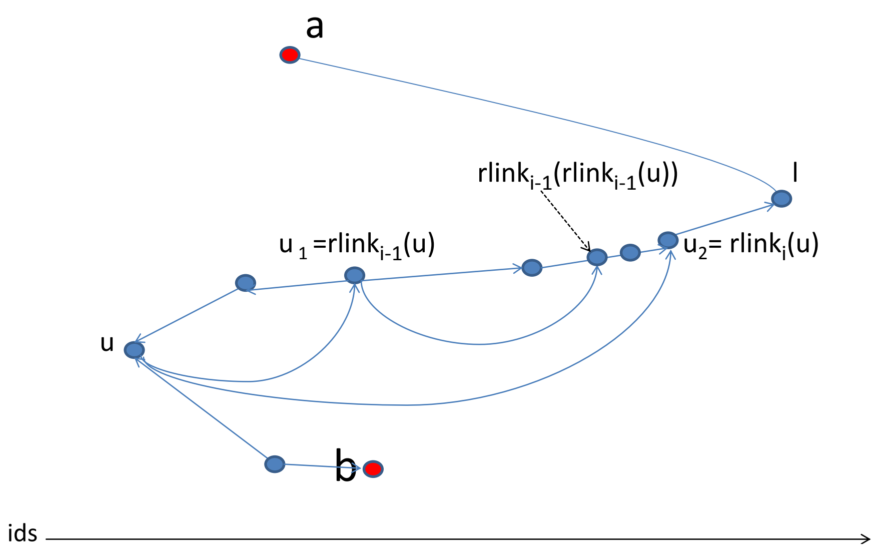


Figure 3.4: With the help of $rlink_i(rlink_i(u)))$, $rlink_i(u))$ is reached from u and integrated into the path from u to l.

$(rlink_{i-1}(u)).Nold$. In case $rlink_{i-1}(rlink_{i-1}(u))$ and u have been neighbors of $rlink_{i-1}(u)$ already at t_0, then at most at timestep $t_1 = 4log(n) + t_0$ during the probing procedure of u node $rlink_{i-1}(u)$ will have been reached and $rlink_{i-1}(rlink_{i-1}(u))$ will have been introduced to u according to the action $(rlink_{i-1}(u)).continueprobingright$ or $(rlink_{i-1}(u)).continueprobingleft$, where the closest node to the probing goal is sent to u, which in this case is $rlink_{i-1}(rlink_{i-1}(u))$. Since $(u, rlink_{i-1}(u))$ and $(rlink_{i-1}(u), rlink_{i-1}\ (rlink_{i-1}(u)))$ have at least $1/2^{log(n)-(i-1)+1}$ length each, then the new edge $(u, rlink_{i-1}(rlink_{i-1}\ (u)))$ will have length at least $1/2^{log(n)-i+1}$, which means that it is stored as $rlink_i(u)$, if u does not already have one. If u did already have a $rlink_i(u)$ l at t_0, then the following cases can occur. In case $l.id > rlink_{i-1}(rlink_{i-1}(u)).id$ then l is dropped as a link and we are done. Otherwise, $rlink_{i-1}(rlink_{i-1}(u))$ is forwarded to l and again the lemma holds. Note that the correctness of the lemma is not influenced if during this time $rlink_{i-1}(u)$ (or analogously/as a consequence $rlink_{i-1}(rlink_{i-1}(u))$) changes its value. We will argue here why. Suppose $rlink_{i-1}(u) = v$ until some timestep t, and at the same timestep u learns a new node w which qualifies better for $rlink_{i-1}(u)$ and sets $rlink_{i-1}(u) = w$. That means, according to the $linearize$ action, that u sends a $request(w)$ message to v and v sends $v.N$ to w at timestep $t + 1$. At timestep $t + 2$ w receives $v.N$ and either sets $rlink_{i-1}(w) = rlink_{i-1}(v)$ or forwards $rlink_{i-1}(v)$ (i.e. what was $rlink_{i-1}(rlink_{i-1}(u))$) to $rlink_{i-1}(w)$. That happens for all existing right links of w, so all existing right links with id less than $b.id$ are on a path from a to b at time $t+2$. Note that the statement of the lemma for u's link $rlink_i(u)$ is also not influenced, since u would still have its $rlink_i(u)$ link on a linear path from a to b.

That is because in order for the value of $rlink_i(u)$ to be changed to $rlink_{i-1}(w)$ either $rlink_{i-1}(w)$ was introduced to u before $t+2$, or u learned about $rlink_{i-1}(w)$ from w during the probing process. But in the latter case this cannot happen until $t+2$. That is because at timestep t u learns about w, so it can at earliest send a continueprobing message to w, which will receive this message at $t+1$ and respond at $t+2$ by sending (possibly) $rlink_{i-1}(w)$ to u. Now, let's consider the other case, where $rlink_{i-1}(w)$ was introduced to u before $t+2$. Here, we consider 2 subcases. In the first subcase u has already had another $rlink_i(u)$ when $rlink_{i-1}(w)$ was introduced, so it just changed the value of $rlink_i(u)$ to $rlink_{i-1}(w)$ and forwarded $rlink_{i-1}(w)$ to the previous $rlink_i(u)$, so the lemma still holds. The second subcase is that u's $rlink_i(u)$ was null when it learned about $rlink_{i-1}(w)$. But that means that $t+1 < t_1 + i \cdot h$, so the lemma holds.

□

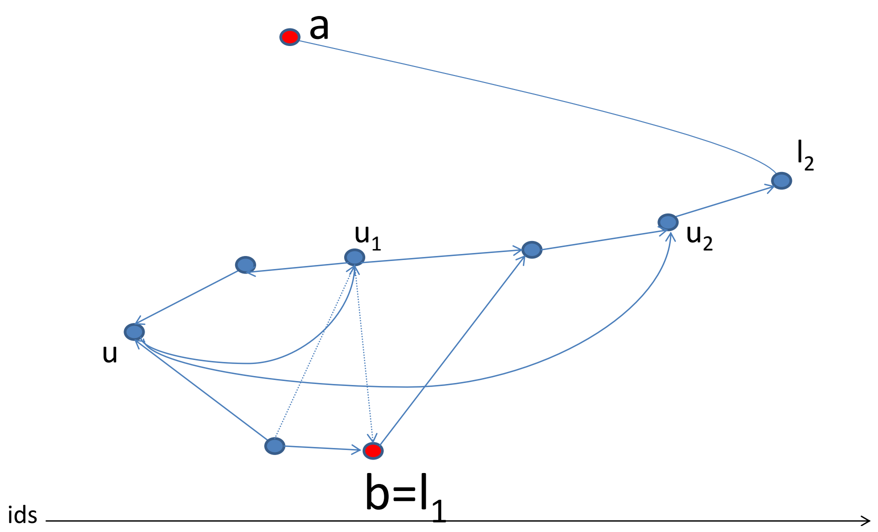


Figure 3.5: (Lemma 3.7)The distance from u to l_1 is halfed every logarithmic number of rounds, since u maintains its a finger more every logarithmic number of rounds (for example u_1) at the path between u and l_1. After $\mathcal{O}(\log^2 n)$ number of rounds l_1 will have a linear path to l_2 and so there will exist a new path from a to b, where the subpath from l_2 to b has only two cornernodes instead of three.

Further we show following lemma.

Lemma 3.7 *At some timestep $t_2 = t_1 + c\log^3 n$, for some constant c, the sorted list (G^{List}) has been formed as a subgraph of the network graph G_{t_2}.*

Proof. Let a and b be two consecutive nodes from the node set V of the network graph, i.e. $a = \max\{w \in V : w.id > b.id\}$. Since the graph is weakly connected, there exists at any time an undirected path from a to b. Let at some timestep $t > t_1$ a (w.l.o.g.) left corner node u on $p_t(a,b)$.

Since u is a corner node, it participates at two linear paths on $p_t(a,b)$, (u,l_1) and (u,l_2), where l_1 and l_2 are the right cornernodes of the subpaths (u,l_1) and (u,l_2) and w.l.o.g. $l_1.id < l_2.id$. Due to Lemma 3.6, at most at some timestep $t_1 + i \cdot h \log n$, u has all its right links of order i (with the link of the i-th order being the largest link that is smaller than $\frac{l_1.id - u.id}{2}$) on (u,l_1) and (u,l_2). That means that $rlink_i(u)$ is both on (u,l_1) and (u,l_2), which means that a new subpath can be formed which is part of $p_{t'}(a,b)$, having $rlink_i(u)$ as a new corner node instead of u. That means that the distance from the cornernode to l_1 has at least halfed after a logarithmic number of rounds. By continuing this argument we get that after $\mathcal{O}(\log^2 n)$ timesteps, the corner node is merged with l_1 and we get a linear path from l_1 to l_2 (Figure 3.2). So we have that for every set of three consecutive corner nodes, after $\mathcal{O}(\log^2 n)$ timesteps only at most two of them are left. So every $\mathcal{O}(\log^2 n)$ timesteps a path from a to b can be constructed, such that the number of corner nodes is reduced by at least 1/3 every time. That means that after $\mathcal{O}(\log^3 n)$ timesteps there is a path between a and b with no corner nodes, in other words there is a direct connection from a to b (or from b to a), and thus, $\mathcal{O}(\log n)$ timesteps later (at latest when a new probing process is conducted) also a direct connection to a from b (or to b from a). Since that happens for every pair of consecutive nodes in V, it means that at most at some timestep $t_2 = t_1 + c \log^3 n$, for some constant c, the sorted list has been formed as a subgraph of the network graph G_{t_2}. □

Phase 2: From the Sorted List to Fast-Re-Chord

Due to Lemma 3.6 we know that at most $\mathcal{O}(\log n)$ timesteps after t_2, each node u on the sorted list maintains its right (reps. left) fingers of order i such that $u.id + 1/2^{\lceil log(n) \rceil - i + 1} < b.id$, with b being the rightmost node in the list, i.e. with the maximal id (resp. $u.id - 1/2^{\lceil log(n) \rceil - i + 1} > a.id$, with a being the leftmost node, i.e. with the minimal id).

Next we show that the ring is formed after $\mathcal{O}(\log n)$ as a subgraph. The ring is the sorted list graph with two additional edges: the ring edge from the node with the maximum value to the node with the minimum value and vice versa.

Lemma 3.8 *The ring is formed as a subgraph of the network graph at timestep $t_2 + d \log n$, for some constant d.*

Proof. Since b is the rightmost node, it has no right neighbor, so it will set its $b.lring$ pointer to the maximum node in $b.N$. During the $linearize()$ action, it will request from $b.lring$ to send its neighbor with the smallest id to b by sending a $(b, leftring)$ message. That procedure happens until the leftmost node (a) is reached. Since the fingers within the range (a,b) are maintained by each node, as described above, in each hop the distance to a is at least halfed, until a is within the direct neighborhood of $u.lring$ and will be returned to u. So a is reached at most after $d \log n$ timesteps after t_2. Due to an analogous argument, b is reached within $d \log n$ timesteps from a and the ring is formed. □

Lemma 3.9 *At timestep $t_3 = t_2 + c' \log n$ a node u is connected to its correct right and left $u.mylogn$ direct neighbors, i.e. $u.rDirect = rDirect(u)$ and $u.lDirect = lDirect(u)$, for a sufficiently large constant c'.*

Proof. W.l.o.g. we consider only $u.lDirect$. Once a new probing starts u will start propagate the probing message to the direct right neighbor and that continues until all $u.mylogn$ right neighbors are reached. Since $u.mylogn = O(\log n)$ this happens in a logarithmic number of timesteps. □

Lemma 3.10 *At timestep $t_4 = t_3 + f \log n \cdot i$ each node u is connected to its correct right (and left) link of order i (i.e. $rlink_i(u) = rfinger_j(u) = u.rfinger_j$, for some $u.first \leq j \leq u.mylogn$), for a sufficiently large constant f.*

Proof.

W.l.o.g. we consider the right links. We will show this through induction.

Induction base:

Note that $rlink_1(u)$ is the first finger from the right with distance at least $1/n$ to u. The next probing message sent after t_3 needs only to traverse the linear path between u and the first node in the sorted list being further away than $1/n$ and qualifies for a finger, that means it is at most $\mathcal{O}(1/n)$ further away. Note that the maximum number of nodes lying between an interval of length $1/n$ is $\mathcal{O}(\log n)$ w.h.p. (as shown in Lemma 3.2). So, there are at most logarithmic number of nodes to be traversed through the probing process. So, all in all, after $t_3 + O(\log n)$ steps, all nodes maintain all their correct links of order 1.

Inductive step:

Due to the induction hypothesis we have that at timestep $t_3 + f \log \cdot i$ each node is connected to its correct right link of order i. Now, let us show this statement for $i + 1$. If u is already connected to $rlink_{i+1}(u)$ (its correct right link of order $i + 1$) we are done. If not, then at latest during the next probing $rlink_i(u)$ is reached within $80 \log n$ steps. That happens, since we know through the induction hypothesis that all links of order $\leq i$ are maintained by all nodes after $t_3 + f \log n \cdot i$ steps, and thus we know that due to the same arguments as in Lemma 3.4, the $rlink_i(u)$ is reached within $80 \log n$ steps w.h.p.. As a consequence $rlink_i(rlink_i(u))$ is introduced to u as a next probing step. From this point on, as all links of order $\leq i$ are maintained by all nodes, the distance to $rlink_{i+1}(u)$ is halfed at each probing hop, so after at most additional $2 \log n$ rounds $rlink_{i+1}(u)$ is reached.

□

Lemma 3.11 *Redundant explicit edges disappear after $\mathcal{O}(\log n)$ steps.*

Proof. Here we show that explicit edges that do not belong to $G^{Fast-Re-Chord}$ disappear over time, so that the set of explicit edges forms the Fast-Re-Chord graph.

So let's consider the timestep t_4 when the graph of the explicit edges G forms a superset of $G^{Fast-Re-Chord}$. Let an explicit edge $(u, v) \in G$, which is not in $G^{Fast-Re-Chord}$. That means that v is not a correct finger of u (or does not belong in $u.N$ in any way) and will be forwarded throught the $linearize$ action to the closest node of $u.N$, w. Since u maintains all its correct fingers, $distance(w, v) \leq \frac{1}{2} distance(u, v)$. Since this happens continuously the distance of the edge to v is halfed in each step, thus after $\mathcal{O}(\log n)$ steps v will be connected with its direct neighbor and the redundant edge no longer exists. □

That finishes the proof for Theorem 3.5.

3.3 Closure

Theorem 3.12 *If the graph of the explicit edges forms $G^{Fast-Re-Chord}$ at time t, then it stills does that at $t' > t$ w.h.p. if no node joins/leaves the system.*

Proof. The actions conducted periodically are the $linearize()$ and the probing actions. Since the nodes maintain their correct neighborhoods (due to the fact that no external dynamics occur), the

$linearize()$ action does not alter the explicit graph. The only way the probing action can add a new explicit edge is if it fails. However due to Lemma 3.4 this does not happen w.h.p. □

3.4 Message Complexity

Theorem 3.13 *The nodes in the network send (and thus receive) $\mathcal{O}(n^2 \log^4 n)$ messages (of logarithmic size) during the self-stabilization.*

Proof. The self-stabilization process takes $\mathcal{O}(\log^3 n)$ steps. In each step each node can send in the worst case all of its neighboorhood ($\mathcal{O}(\log n)$) to all other nodes in the graph $(n - 1)$. Thus the Theorem follows. □

Theorem 3.14 *In the stable state, a constant number of messages are sent/received per node per step on average.*

Proof. The only messages that are sent in a stable state are the messages associated with probing. Obviously a node sends only one $probingl$, $probingr$ message in each timestep. Also to each node exactly one $continueprobingright$ and $continueprobingleft$ message is sent in each timestep from some other node. Thus each node sends and receives on average a constant number of messages per timestep. □

4 Discussion and Extensions

Note that, as far as the message complexity is concerned, a lower bound for the algorithm (as well as for any linearization-based algorithm) is $\Omega(n^2)$ (messages of logarithmic size). That is because at the worst case the intial graph could be a clique, in which case each node has to forward $\mathcal{O}(n)$ nodes to its neighbors only in the first timestep.

Concerning the time complexity, the only known approaches to achieve better time ($\mathcal{O}(\log n)$ number of timesteps) for self-stabilizing topologies are approaches based on all-to-all introduction, where nodes introduce all their neighbors to all their neighbors at each timestep. However these approaches are very inefficient in terms of message complexity, since at least $\Omega(n^3)$ messages of logarithmic size (or $\Omega(n^3 \log n)$ communication complexity) are needed for the convergence. The message complexity in the stable state is also large.

So the technique presented in this chapter is the first to achieve polylogarithmic self-stabilization time without using the message expensive all-to-all introduction procedure, at least in the field of topological self-stabilization.

Note the protocol presented in this chapter is applicable for the framework presented in the previous chapter, i.e.: the conditions required are met here. That means that the algorithmic framework of chapter 2 can be applied to this protocol, which will result to a protocol that solves $\mathcal{FDP}$ and reaches the *Fast-Re-Chord* topology.

Chapter 4

Introducing Heterogeneity

An efficient self-stabilizing algorithm for a heterogeneous storage system

In the previous chapter we considered how to self-stabilizingly construct and maintain an efficient distributed hash table. In this chapter we will add another aspect in this issue, the heterogeneity. Heterogeneity can imply many things, but usually in distributed systems we mean heterogeneity at node level: i.e. nodes can have different capacities, bandwidths, and reliabilities. Here we consider the nodes's capacity. As a matter of fact, in this particular chapter we consider the problem of managing a dynamic heterogeneous storage system in a distributed way so that the amount of data assigned to a host in that system is related to its capacity.

Three central problems have to be solved for this: (1) organizing the hosts in an overlay network with low degree and diameter so that one can efficiently check the correct distribution of the data and route between any two hosts, (2) distributing the data among the hosts so that the distribution respects the capacities of the hosts and can easily be adapted as the set of hosts or their capacities change and (3) doing this in an efficient way, regarding the timesteps as well as the messages required. We present a distributed protocol for these problems that is self-stabilizing and that does not need any global knowledge about the system such as the number of nodes or the overall capacity of the system. Prior to this work no solution was known satisfying these properties.

The work presented in this chapter is based on the following paper:

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler: CONE-DHT: A Distributed Self-Stabilizing Algorithm for a Heterogeneous Storage System. DISC 2013

In fact, it improves the above paper in certain ascepts, since the presented protocol is a new version of the original one, which enables fast self-stabilization, and lower message complexity as well as fairly distributed routing overhead. So part of the protocol as well as the time-complexity proof presented in this chapter are completely new and are firstly presented in this thesis.

1 Heterogeneous Storage Systems

Many data management strategies have already been proposed for distributed storage systems. If all hosts have the same capacity, then a well-known approach called *consistent hashing* can be used to manage the data [23]. In consistent hashing, the data elements are hashed to points in $[0, 1)$, the hosts are mapped to disjoint intervals in $[0, 1)$, and a host stores all data elements that are hashed to points in its interval. 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 [58]. These strategies have been realized in various DHTs including CAN [24], Pastry [25] and Chord [2]. However, all of these approaches assume hosts of uniform capacity, despite the fact that in P2P systems the peers can be highly heterogeneous.

In a heterogeneous setting, each host (or node) u has its specific capacity $c(u)$ and the goal considered in this chapter 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., [43, 41, 44]). In [43] 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 [44]. In [40] 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. [20] 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 [45, 46, 50, 49, 47, 48] or assume a central server that handles the mapping of data elements to a set of hosts [19, 22, 21]. The only solution proposed so far where this is not the case is the approach by Schindelhauer and Schomaker [19], 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.

Heterogeneity can be considered in many ways in a DHT. Another kind is bandwidth heterogeneity. A self-stabilizing protocol to that problem was given indeed in [31], where the authors give a variation of a self-stabilizing skip-graph protocol in order to handle routing in a network with nodes with different bandwidths. Heterogeneity can also affect other aspects, like the reliability in a system where churn is considered. However, self-stabilizing protocol for that setting was not known until now.

The problem of self-stabilizingly managing heterogeneous hosts (in terms of capacity) in a DHT was not considered until now. So, to the best of our knowledge this is the first self-stabilizing approach for a distributed heterogeneous storage system.

More specifically, for a solution for a self-stabilizing distributed heterogeneous storage system the following is derived:

- *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.
- *Time efficiency*: The network does not need more than polylogarithmic number of rounds in order to converge to the stable state
- *Routing efficiency*: There is a routing strategy that allows efficient routing in at most $\mathcal{O}(\log n)$ hops, and the routing message overhead is equally distributed among the nodes
- *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*).

We futher present a protocol that matches these criteria.

2 The *CONE*-DHT

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 [19] (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 main result:

Theorem 2.1 *There is a self-stabilizing algorithm for maintaining a heterogeneous storage system that stabilizes in $\mathcal{O}(\log^4 n)$ number of rounds and achieves fair load-balancing, space efficiency and routing efficiency, while each node has a degree of $\mathcal{O}(\log n)$ w.h.p. The data operations can be handled in $\mathcal{O}(\log n)$ time in a stable system, and if a node joins or leaves a stable system or changes its capacity, it takes at most $\mathcal{O}(\log^2 n)$ structural changes, i.e., edges that are created or deleted, until the system stabilizes again.*

2.1 The original CONE-Hashing

Before we present our solution, we first give some more details on the original CONE-Hashing [19] our approach is based on. In [19] 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 u a hash value $u.id$. 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 [19] each node has a capacity function $c(u)(g(x))$, which determines which data is assigned to the node. A node is *responsible* for those elements d with $c(u)(g(d)) = \min_{v \in V}\{c(v)(g(d))\}$, i.e. d is assigned to u. We denote by $R(u) = \{x \in [0,1) : c(u)(x) = \min_{v \in V}\{c(v)(x)\}\}$ the *responsibility range* of u Note that $R(u)$ can consist of several intervals in $[0,1)$. In the original paper [19], the authors considered two special cases of capacity functions, one of linear form $C_u^{lin}(x)$and of logarithmic form $C_u^{log}(x)$(these functions we will discuss later on). For these capacity functions the following results were shown by the authors [19]:

Theorem 2.2 *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 capacity functions* $C_u^{lin}(x)$ *and with probability* $\frac{c(u)}{\sum_{v \in V} c(v)}$ *for logarithmic capacity functions* $C_u^{log}(x)$. *Thus in expectation fair load balancing can be achieved by using a logarithmic capacity function* $C_u^{log}(x)$.

The CONE-Hashing 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.

Moreover, the authors showed that the fragmentation is relatively small for the logarithmic capacity function, with each node having in expectation a logarithmic number of intervals it is responsible for. In the case of the linear function, it can be shown that this number is only constant in expectation.

In [19] 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. Further the following challenges have to be solved. We need 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 should also support an efficient *Search(d)* algorithm, i.e. for a *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)*, *CapacityChange(v)* operation should not lead to a high amount of data movements, (i.e. not 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). All these challenges will be solved by our CONE-DHT. In the CONE-DHT, the same sets of capacity functions can be used as discussed here, and thus our system can inherit the same properties.

2.2 The *CONE*-DHT

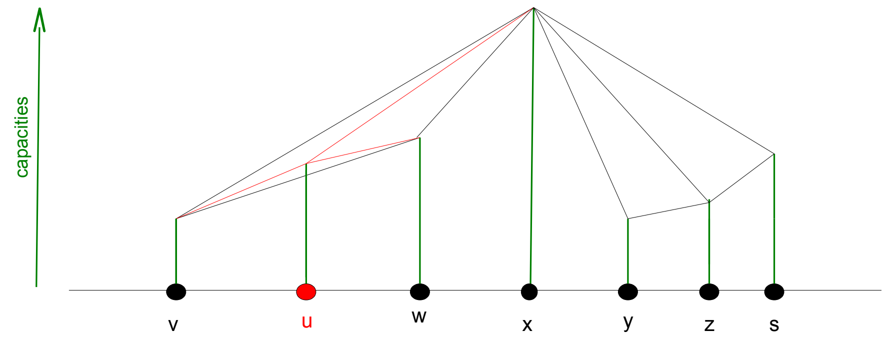
In order to construct a heterogeneous storage network in the distributed case, we have to deal with the challenges mentioned above. For that, we introduce the *CONE*-graph, which is an overlay network that combines elements from CONE-DHT and the *Fast- Re-Chord* protocol, and, as we show, can support efficiently a heterogeneous storage system.

The network layer

We define the *CONE* graph, our goal topology, as a graph $G^{CONE} = (V, E^{CONE})$, with V being the hosts of our storage system.

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

Figure 4.1: In this example, the size of the capacity of a node is symbolized by the height of its green column (i.e. the larger the capacity the higher the column). So, for example in this case u is aware of v, w and x. In fact, $S^+(u) = \{w, x\}, S^-(u) = \emptyset, P^+(u) = \emptyset, P^-(u) = \{v\}$



- $succ_1^+(u) = argmin\{v.id : v.id > u.id \wedge c(v) > c(u)\}$ 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: $pred_1^+(u) = argmax\{v.id : v.id < u.id \wedge c(v) > c(u)\}$ 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 $u = succ_1^+(v)$.

Now we can define the edge-set of a node in G^{CONE}.

Definition 2.3 $(u, v) \in E^{CONE}$ *iff* $v \in S^+(u) \cup P^+(u) \cup S^-(u) \cup P^-(u) \cup E^{Re-Chord}$

We define also the neighborhood set of u as $N_u = \{v : (u, v) \in E^{CONE}\}$. We also consider $S(u) = S^+(u) \cup S^-(u)$ and $P(u) = P^+(u) \cup P^-(u)$. In other words, v maintains connections to each node u, if there does not exist another node with larger capacity than u between v and u (see Figure 4.1), or if (u, v) is a $Fast - Re - Chord$ edge. 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.. The capacity functions we use are $C_u^{lin}(x) = \frac{1}{c(u)}|x -$

$u.id|$ and $C_u^{log}(x) = \frac{1}{c(u)}(-log(1-|x-u.id|)$ (Note that the algorithm works for $C_u^{lin}(x) = \frac{1}{c(u)}((x-u.id) \mod 1)$ and $C_u^{log}(x) = \frac{1}{c(u)}(-log((1-(x-u.id)) \mod 1)$, with slight modifications).

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. 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* graph has been established, this node is aware of the responsible node for this data item. We call the interval between $u.id$ and the hash value of u's closest right node I_u. We say that u is *supervising* I_u. We show the following theorem.

Theorem 2.4 *In G^{CONE} 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 $R(v) \cap I_u \neq \emptyset$ are in $S^+(u) \cup P^+(u) \cup S^-(u) \cup P^-(u)$. W.l.o.g. let us consider only the case of $S^+(u), S^-(u)$. Indeed, there cannot be a node at the right of u ($u.id < t.id$) that has a responsible interval in u's supervising interval and that is not in $S^+(u)$ or $S^-(u)$.We will prove it by contradiction. Let t be such a node. For t not to be in $S^-(u)$ or $S^+(u)$ there must be at least one node v larger (in terms of capacity) than t, which is closer to u than t ($u.id < v.id < t.id$). Then $\forall x < v.id$ it holds that $v.id < t.id \implies x - v.id > x - t.id \implies |x - v.id| < |x - t.id|$ (since both sides are negative) $\implies f(x - v.id) < f(x - t.id)$, for $f(z) = z$ and $f(z) = -log(1 - z)$, for the right side of f. Moreover, since $c(v) > c(t)$ we have $\frac{1}{c(v)}f(x - v.id) < \frac{1}{c(v)}f(x - t.id) \implies C_v(x) < C_t(x)$, so v dominates t for $x < v.id$. And since $u.id < v.id$, it cannot be that t is responsible for an interval in I_u, since in that region t is dominated (at least) by v. This contradicts the hypothesis and the proof is completed. □

So, the nodes store their data in the following way. If a node u has a data item that falls into one of its responsible intervals, it stores in addition to this item a reference to the node v that is the closest from the left to this interval. Moreover, the sub-interval u thinks it is responsible for (in which the data item falls) is also stored (as described in the next section, when the node's internal variables are presented). In case the data item is not stored at the correct node, v can resolve the conflict when contacted by u.

Now we can discuss the functionality of the data operations. A node has operations for inserting, deleting and searching a datum in the CONE-network.

Let us focus on **searching** a data item. As shown above, it suffices to search for the left closest node to the data item's hash value. We do this by using greedy routing. Greedy routing in the *CONE*-network works as follows: If a search request wants to reach some position pos in $[0, 1)$, and the request is currently at node u, then u forwards $search(pos)$ to the node v in N_u that is closest to pos, until the closest node at the left of pos is reached. Then this node will forward the request to the responsible node.

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. Note that the **deletion** of a data item can be realized in the same way, only that when the item is found, it is also deleted from the responsible node.

Inserting an item follows a similar procedure, with the difference that when the responsible node is found, the data item is stored by it.

Moreover, the network handles efficiently structural operations, such as the joining and leaving of a node in the network, or the change of the capacity of a node. Since this handling falls into the analysis of the self-stabilization algorithm, we will discuss the network operations in Section 3, where we also formally analyze the algorithm.

It turns out that a single data or network operation (i.e greedy routing) can be realized in a logarithmic number of hops in the *CONE*-network, and this happens due to the structural properties of the network, which we discuss in the next section, where we also show that the degree of the *CONE*-network is logarithmic.

2.3 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 $\mathcal{O}(\log n + |\text{amount of data stored in a node}|)$ w.h.p..

Theorem 2.5 *The degree of a node in a stable CONE network is $\mathcal{O}(\log n)$ w.h.p.*

Proof. First we show following lemma:

Lemma 2.6 *In a stable CONE network for each $u \in V$, $|S^+(u)|$ and $|P^+(u)|$ in $\mathcal{O}(\log n)$ w.h.p.*

Proof. For an arbitrary $u \in V$ let $W = \{w_1, w_2 \cdots w_k\} \cup \{w_0 = u\} = S^+(u) \cup \{u\}$ and let W be sorted by ids in ascending order, such that $w_i.id < w_{i+1}.id$ for all $1 \leq i < k$. Furthermore, let $\hat{W}(w_i) = \{w \in V : w.id > w_i.id \wedge c(w) > c(w_i)\}$ be the set of all nodes with larger ids and larger capacities than w_i. So, the determination of W 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 $w_i = \hat{w_j}$ *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 $|\hat{W}(w_0)| = m = \mathcal{O}(\log n)$. Since the id/position for each node 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 $\log m$ good choices the remaining set $\hat{W}(w_i)$ is the empty set. Thus there can not be more than $\log m$ good choices in any sequence of choices. So, what we have now is a random experiment, that is described by the random variable k, that is equal to the number choices we must make, until we managed to have made $\log m$ good ones. Then the random variable k follows the negative binomial distribution. In order to bound the value of k from above we apply the following tail bound for negative binomial distributed random variables shown in [38], derived by using a Chernoff bound:

Claim 2.7 *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 $\delta \in [0, 1]$ and set $l = \frac{s}{(1-\delta)p}$. Then $Pr[Y > l] \leq exp(\frac{-\delta^2 s}{3(1-\delta)})$*

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

Lemma 2.8 *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 x being in an interval $S^-(u)$ it holds $u = pred_1^+(x)$. But since each node has (at most) one $pred_1^+$, the sum over all $S^-(v)$, $\forall v \in V$ must be (at most) n. So we have $\sum_{\forall v \in V} S^-(v) = n$, so $E[\sum_{\forall v \in V} S^-(v)] = n$ $\Rightarrow \sum_{\forall v \in V} E[S^-(v)] = n$. That means for a node u, $E[S^-(u)] = 1$.

Now we consider the second part of the statement. Let w 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 $S^-(u)$ (expect w) must be in $S^+(w)$. Let us assume a node x is in $S^+(w)$ but not in $S^-(u)$, then there must be another node $y : u.id < y.id < x.id$ and $c(y) > c(x)$, such that $y \in S^-(u)$. But then y would be also in $S^+(w)$ instead of x. So, we contradicted this scenario. As a consequence $S^-(u)/\{w\} \subset S^+(w)$, but we already shown that $|S^+(w)| < \log n$ w.h.p., from which follows that $|S^-(u)| < \log n$ w.h.p.. □

We already know from the analysis of the Fast-Re-Chord network, that the number of edges a node maintains is $\mathcal{O}(\log n)$. So, combining that fact with the results shown here proves the theorem.

□

Additionally to the nodes in $u.N$ that lead to the degree of $\mathcal{O}(\log n)$ outgoing edges w.h.p. a node u only stores references about the closest nodes left to 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. Thus the storage only has a logarithmic overhead for the topology information and the following theorem follows immediately.

Theorem 2.9 *In a stable CONE network each node stores at most $\mathcal{O}(\log n + |$amount of data stored in a node$|)$ information w.h.p.*

One last property about the incoming edges of a node.

Theorem 2.10 *The number of incoming explicit edges to a node u in the stable state is $\mathcal{O}(\log n)$ w.h.p..*

Proof. Let's first consider the nodes having u as a right finger. In order for a node v to have u as a right link of level i, its must be that there is no other node between u and $v.id + 1/2^{\lceil log(n) \rceil - i + 1}$. Obviously the average number of the nodes having this property is 1. Since the number of levels, for which links exist is $\mathcal{O}(\log n)$ w.h.p., by simply applying a Chernoff bound we get that the number of nodes having u as a right finger is $\mathcal{O}(\log n)$. Symmetrically, the number of nodes having u as a left finger is $\mathcal{O}(\log n)$. We also know that the nodes having u as a direct neighbor is $\mathcal{O}(\log n)$ and the proof is completed. □

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.

3 Self-Stabilization Process

3.1 Formal Problem Definition and Notation

Now we define the problem we solve in this chapter. We provide a protocol *P* that reaches the overlay problem *CONE* and is topologically self-stabilizing.

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

- $u.S^+ = \{v \in N_u : v.id > u.id \wedge c(v) > c(u) \wedge \forall w \in N_u : v.id > w.id > u.id \implies c(v) > c(w)\}$
- $u.succ_1^+ = \text{argmin}\,\{v.id : v \in u.S^+\}$: The first node to the right with a larger capacity than u
- $u.P^+ = \{v \in N_u : v.id < u.id \wedge c(v) > c(u) \wedge \forall w \in N_u : v.id < w.id < u.id \implies c(v) > c(w)\}$
- $u.pred_1^+ = \text{argmax}\,\{v.id : v \in u.P^+\}$: The first node to the left with a larger capacity than u
- $u.S^- = \{v \in N_u : v.id > u.id \wedge c(v) < c(u) \wedge \forall w \in N_u : v.id > w.id > u.id \implies c(v) > c(w)\}$
- $u.P^- = \{v \in N_u : v.id < u.id \wedge c(v) < c(u) \wedge \forall w \in N_u : v.id < w.id < u.id \implies c(v) > c(w)\}$
- $u.S^* = \{u.S^- \cup \{u.succ_1^+\}\}$: the set of right neighbors that u communicates with. We assume that the nodes are stored in ascending order so that $(u.S^*[i]).id < (u.S^*[i+1]).id$. If $|u.S^*| = k$, then $u.S^*$, $u.S^*[k] = u.succ_1^+$.
- $u.P^* = \{u.P^- \cup \{u.pred_1^+\}\}$: the set of left neighbors that u communicates with. We assume that the nodes are stored in descending order so that $(u.P^*[i]).id > (u.P^*[i+1]).id$ If $|u.P^*| = k$, 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

With $u.S$ (rep. $u.P$) we also denote the union of $u.S^+$ and $u.S^-$ (rep. $u.P^+$ and $u.P^-$). Additionally each node stores the following variables :

- τ: the timer predicate that is periodically true
- $u.I_u$: the interval between u and the successor of u. u is supervising $u.I_u$.
- m: the message in Ch_u that now received by the node.

Additional to that, each node maintains the same variable set used in the fast Fast-Re-Chord protocol:

- $u.mylogn$: The estimation of $\log(n)$.
- $u.firstf$: The order of the first finger, computed respectively to the "closest" node to u from u's neighborhood
- $u.rfinger_i, \forall i$ with $u.firstf \leq i \leq u.mylogn$: The right finger of node u of order i.

- $u.lfinger_i, \forall i$ with $u.firstf \leq i \leq u.mylogn$: The left finger of node u of order i.
- $u.rDirect$: The set of nodes in u's neighborhood, being "closest" to u from the right.
- $u.lDirect$: The set of nodes in u's neighborhood, being "closest" to u from the left.
- $u.rring$: The right ring edge of u.
- $u.lring$: The left ring edge of u.
- $u.Pr$: The nodes used for supporting the probing process of u.
- $u.nextr$ A pointer that points to the next node of the right probing process.
- $u.nextl$ A pointer that points to the next node of the left probing process.
- $u.rprobcounter$ Counts the number of steps the right probing process currently lasts
- $u.lprobcounter$ Counts the number of steps the left probing process currently lasts

$u.Rfingers$ (resp. $u.Lfingers$) is also used to denote the set of all right fingers (resp. left fingers). Moreover, $u.N$ is the neighborhood of u, which is includes all the nodes stored in u, i.e. the nodes in $u.Rfingers$,$u.Lfingers$, $u.rDirect$, $u.lDirect$, $u.Pr$,$u.S$,$u.P$ as well as $u.rring$ and $u.lring$. Note also that sometimes instead of a variable $u.var$ in the pseudocode we also write just var, since it is clear that var refers to the node u, which is the node conducting the protocol described at the $pseudocode$.

Definition 3.1 *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* $w.id < u.id$ *or* $c_w < c(u)$ *or* $u.id < v.id < w.id$ *and* $c(v) > c_w$ *for any* $v \in N_u$.

Now we can describe the topologies in the initial states and in the legal stable state. Let $IT = \{G^{IT} = (V, E_{IT} = E_e^{IT} \cup E_i^{IT}) : G^{IT} \text{ is weakly connected}\}$ and let $CONE = \{G^C = (V, E^C)\}$, such that for E^C the following conditions hold: (1) $E^C = E_e - \{(u,v) : v \in u.DS\}$, (2) E^C is in a valid state and (3) $E^C = E^{CONE}$.

Note that we assume E_e to be a multiset, i.e in E^C an edge (u,v) might still exists, although $v \in u.DS$ if e.g. $v \in u.S^+$. Further note that, in case the network has stabilized to a *CONE*-network, it holds for every node that $u.S^+ = S^+(u), u.P^+ = P^+(u), u.S^- = S^-(u)$ and $u.P^- = P^-(u)$.

3.2 Algorithm

In this section we give a description of the the distributed 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 the periodic actions each node performs a consistency check of its internal variables, i.e. are all variables valid according to Definition 3.1. If some variables are invalid, the nodes causing this invalidity are delegated.

Furthermore in the periodic actions the same actions are performed as in the *Fast-Re-chord* protocol, in order to form an underlying *Fast-Re-chord* graph. Moreover, each node periodically introduces itself to its successor and predecessor $u.S^*[1]$ and $u.P^*[1]$ by a lin message. u also introduces the nodes $u.succ_1^+$ and $u.pred_1^+$ to each other by messages of type lin. By this a *triangulation* is formed by edges $(u, u.pred_1^+), (u, u.succ_1^+), (u.succ_1^+, u.pred_1^+)$ (see Figure 4.1). 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 $(u.S^+ \cup \{u\}, list - update)$ (resp. $(u.P^+ \cup \{u\}, list - update)$ to v. Also a node periodically sends 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 $([a, b], u, check - interval)$.

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 (v, lin) u checks whether v has to be included in it's internal variables. If u doesn't store v, v is delegated. If u receives a message $(list, list - update)$, u checks whether the ids 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. $u.succ_1^+$). There are also message types that are necessary for the data management.

If u receives a message $([a, b], v, check-interval)$ it checks whether v is in $u.S^+$ or $u.P^+$ or has to be included in $u.N$, 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 $(IntervalSet, update - interval)$ 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 $(IntervalSet, update - interval,)$ it forwards all data in intervals in $IntervalSet$ to the corresponding references by a message $(data, forward - data)$. 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 $(data, u, store - data)$ 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.

4 The Protocol

Here we give the pseudocode for the protocol executed by each node.

In Algorithm 18 we can see the periodic action of a node u. Probing is initiated if it is currently not happening. Moreover, the consistency of the ring edges is checked. Also a consistency check is included, where all list $u.P^+, u.S^+, u.S^-, u.P^-$ are checked and invalid nodes are delegated like in the ListUpdate/BuildTriangle operation. Furthermore each node sends its lists $u.P^+, u.S^+$ to the next smaller nodes and $u.S^+$ to its direct left neighbor. In the BuildTriangle() u introduces itself to its neighbors and neighbored nodes in $u.P^*$ and $u.S^*$ and $u.pred_1^+$ and $u.succ_1^+$ to each other and checks whether the information in $u.DS$ is still up to date.

Algorithm 18 PERIODIC ACTIONS OF NODE U

```
true$\rightarrow$
u.lprobcounter:=u.lprobcounter+1
u.rprobcounter:=u.lprobcounter+1
if $u.rprobcounter > 82u.mylogn$ then                ▷ If the probing ended, start probing
    u.probingr($\min\{w \in u.N : w.id > u.id\}, u$)
    u.rprobcounter=0
if $u.lprobcounter > 82u.mylogn$ then
    u.probingl($\max\{w \in u.N : w.id < u.id\}, u$)
    u.lprobcounter=0
if $u.id \geq w.id : \forall w \in u.N$ then                ▷ ring edge checks
    send message(u,requestleftring) to u.lring
else
    linearize(u.lring)
    u.lring=null
if $u.id \leq w.id : \forall w \in u.N$ then
    send message(u,requestrightring) to u.rring
else
    linearize(u.rring)
    u.rring=null
Do Consistency check for $u.P^+$, $u.S^+$, $u.S^-$, $u.P^-$    ▷ trivial check, pseudocode omitted here
readability
if u.counter=u.mylogn then
    send message($u.P^+ \cup \{u\}$,list-update) to $u.S^-[1]$
    send message($u.S^+ \cup \{u\}$,list-update) to $u.P^-[1]$
    BuildTriangle()
    CheckDataIntervals()
    u.counter=0
u.counter=u.counter+1
```

Algorithm 19 PERIODIC ACTIONS OF NODE U (continued)

```
message $m \in p.C \rightarrow$                ▷ actions upon receiving a message
if m.type=lin then
    linearize(m.id)
else if m.type=probingr then
    probingr(m.id,m.sender)
else if m.type=probingl then
    probingl(m.id,m.sender)
else if m.type=requestprobingright then
    continueprobingright(m.sender,stage)
else if m.type=requestprobingleft then
    continueprobingleft(m.sender,stage)
else if m.type=requestrightring then
    rightring(m.id)
else if m.type=requestleftring then
    leftring(m.id)
else if m.type=requestneib then
    requestneib(m.id)
else if m.type=list-update then
    ListUpdate(List)
else if m.type=check-interval then
    CheckInterval([a,b],m.id)
else if m.type=update-interval then
    UpdateInterval(IntervalSet)
else if m.type=forward-data then
    ForwardData(data,boolean)
else if m.type=store-data then
    StoreData(data,interval)
linearize()
validitycheck() ▷ Check whether the nodes in $u.N$ are within the boundaries they are supposed to,
else linearize them (is a trivial procedure and omitted here)
```

Algorithm 20 U.LINEARIZE(V)

```
if $v \notin u.N \vee v = \emptyset$ then
    $u.Nold = u.N$
    $u.N = u.N \cup \{v\}$
    if $u.id \leq w.id, \forall w \in u.N$ then                ▷ Update ring edges
        u.rring=max$\{w \in u.N\}$
    else
        if $u.rring \neq null$ then
            u.linearize(u.rring)
        u.rring=$null$
    if $u.id \geq w.id, \forall w \in u.N$ then
        u.lring=max $w \in u.N$
    else
        if $u.lring \neq null$ then
            u.linearize(u.lring)
        u.lring=$null$
    $u.mylogn = \max\{k \in \mathbb{N} : \exists w \in u.N : |w.id - u.id| \leq 1/2^k\}$        ▷ estimate log(n)
    $u.firstf = \min\{k \in \mathbb{Z} : \nexists w \in u.N : |w.id - u.id| \leq 1/2^{mylogn-k+1}\}$
    for all $i : u.firstf \leq i \leq mylogn$ do        ▷ compute right and left fingers
        $u.rfinger_i = \text{argmin}_{w \in u.N}\{w.id : w.id \geq v.id + 1/2^{mylogn-i+1} (\text{mod } 1)\}$
        if $u.rfinger_i = null$ then
            $u.rfinger_i = \text{argmin}_{w \in u.N}\{w.id\}$
        $u.lfinger_i = \text{argmax}_{w \in u.N}\{w.id : w.id \leq v.id - 1/2^{mylogn-i+1} (\text{mod } 1)\}$
        if $u.lfinger_i = null$ then
            $u.lfinger_i = \text{argmax}_{w \in u.N}\{w.id\}$
    u.Rfingers=$\bigcup_{\forall firstf \leq i \leq mylogn} u.rfinger_i$
    u.Lfingers=$\bigcup_{\forall firstf \leq i \leq mylogn} u.lfinger_i$
                                        ▷ 6mylogn nodes closest to $u$ from the right and the left
    $u.rDirect = \{w \in u.N : |\{v \in u.N : x \leq y$, where $x = \min\{v.id - u.id, 1 - u.id + v.id\}, y = \min\{w.id - u.id, 1 - u.id + w.id\} \wedge x, y > 0\}| \leq 6mylogn\}$
    $u.rDirect = \{w \in u.N : |\{v \in u.N : x \leq y$, where $x = \min\{u.id - v.id, 1 - v.id + u.id\}, y = \min\{u.id - w.id, 1 - w.id + u.id\} \wedge x, y > 0\}| \leq 6mylogn\}$

    $u.succ_1^+ = w \in u.N : w.id > u.id \wedge c(w) > c(u) \wedge (\nexists y : u.id < y.id < w.id \wedge c(y) > c(u))$
    $u.pred_1^+ = w \in u.N : w.id < u.id \wedge c(w) > c(u) \wedge (\nexists y : u.id > y.id > w.id \wedge c(y) > c(u))$
    update $u.S, u.P$
    for all $w \in u.N : u \notin Rfingers \cup Lfingers \cup rDirect \cup lDirect \cup \{rring\} \cup \{lring\} \cup u.Pr \cup u.P \cup u.S$ do
        u.forward(w)                                    ▷ get rid of the rest
    $u.N = Rfingers \cup Lfingers \cup rDirect \cup lDirect \cup \{rring\} \cup \{lring\} \cup u.Pr \cup u.P \cup u.S$
    **if** $u.N \neq u.Nold$ **then**                                   ▷ If v was included in new u.N
        **for all** $w \in u.N$ **do**
            send message (w,lin) to $v$
        send message (v, requestneib) to h=$\{u.Nold.finger_j : v = u.finger_j\}$
    **else**
        u.forward(v)
```

Algorithm 21 U.PROBINGR(V,M.SENDER)

```
  As in Algorithm 10.
```

Algorithm 22 U.PROBINGL(V,M.SENDER)

```
  As in Algorithm 11.
```

Algorithm 23 U.FORWARD(V)

```
                                                         ▷ forward v to the closest neighbor
  As in Algorithm 12.
```

Algorithm 24 U.CONTINUEPROBINGRIGHT(M.SENDER,STAGE)

```
                      ▷ Sends the next probing node to the sender, so that the probing can be continued
  As in Algorithm 13.
```

Algorithm 25 U.CONTINUEPROBINGLEFT(M.SENDER,STAGE)

```
                      ▷ Sends the next probing node to the sender, so that the probing can be continued
  As in Algorithm 14.
```

Algorithm 26 U.RIGHTRING(V)

```
  As in Algorithm 15.
```

Algorithm 27 U.LEFTRING(V)

```
  As in Algorithm 16.
```

Algorithm 28 U.REQUESTNEIB(V)

```
  As in Algorithm 17.
```

```
  **function** BUILDTRIANGLE
    **for all** $w \in u.S^- \cup u.P^- \cup \{u.succ_1^+, u.pred_1^+\}$ **do**
        send message($u$,lin) to $w$
    send message($u.pred_1^+$,lin) to $u.succ_1^+$ and message($u.succ_1^+$,lin) to $u.pred_1^+$
```

```
function ListUpdate(List)
    $LList^+ = z \in List : z.id < u.id \wedge c(z) > c(u)$        ▷ candidates for $u.P^+$
    $LList^- = z \in List : z.id < u.id \wedge c(z) < c(u)$        ▷ candidates for $u.P^-$
    $RList^+ = z \in List : z.id > u.id \wedge c(z) > c(u)$        ▷ candidates for $u.S^+$
    $RList^- = z \in List : z.id > u.id \wedge c(z) < c(u)$        ▷ candidates for $u.S^-$
    calculate $P^+_{tmp}$ out of$u.pred^+_1$ 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 message(u,lin) to $z$
        $u.P^+ = P^+_{tmp}$
    calculate $S^+_{tmp}$ out of $u.succ^+_1$ 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 message(u,lin) to $z$
        $u.S^+ = S^+_{tmp}$
    calculate $P^-_{tmp}$ out of $u.P^-$ and $LList^-$
    for all $w \in (u.P^- \cup LList^-) - P^-_{tmp}$ do
        $v^+(w) =$ argmin $\{v.id : v \in P^-_{tmp} \cup \{pred^+_1\} \wedge v.id > w.id\}$
        send message(w,lin) 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
        $v^-(w) =$ argmax $\{v.id : v \in S^-_{tmp} \cup \{succ^+_1\} \wedge v.id < w.id\}$
        send message(w,lin) to $v^-(w)$
    $u.S^- = S^-_{tmp}$
```

A node checks for each interval it is responsible for, if this is really the case.

```
function CheckDataIntervals
    for all u.DS[i] do
        send message ($[a, b] = u.DS[i]$,$u$,check-interval) to $u.DS[i].ref$
```

A node receiving a check-interval message, checks if the node which sent it is really responsible for the interval [a,b].

```
function CheckInterval([a,b],x)
    if $x \notin u.P^{+} \cup u.S^{+} \cup u.S^{-}$ then
        linearize(x)
    IntervalSet := $\emptyset$
    i:=1
    if $a < u.id$ then
        IntervalSet[i]=$[a, u.id] \cap [a, b]$ ▷ 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
    $[c, d] := I_u(x)$ ▷ $I_u(x)$ is the sub-interval of $u.I_u$ for which $x$ is responsible for
    [e,f]:=$([a, b] \cap u.I_u)/I_u(x)$
    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
    if $IntervalSet \neq [a, b]$ then
        send message(update-interval,IntervalSet) to $x$
    for $i = 1 : |IntervalSet|$ do
        if $IntervalSet[i].ref \neq u$ then
            send message(IntervalSet[i],update-interval) to IntervalSet[i].ref
```

By receiving an update-interval message, a node updates the lists of intervals which it is responsible for, and forwards the data in the deleted intervals to another node, who is possibly responsible.

```
function 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 message (data,forward-data) to $[a, b].ref$ ▷ data $u$ seems not to be responsible for or for that the reference changed is deleted
                delete(data)
                linearize([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
```

By receiving a forward-data message, a node checks if it knows which node is responsible for the data it received, and sends a store-data message to it, in the other case it also forwards the data.

```
function FORWARDDATA(data)
    if $data.id \notin u.I_u$ then
        if $data.id \in [u.P^*[1]], u]$ then
            send message (data,forward-data) to $u.P^*[1]$
        else
            send message (data,forward-data) to
            $w : (u.id < w.id < data.id \vee u.id > w.id > data.id) \wedge |data.id - w.id| = \min_{y \in u.P^* \cup u.S^*}\{|data.id - y.id|\}$,
    else
        send message (data,$I_u(v)$,u,store-data) to $v : data.id \in I_u(v)$
```

Storing the data received from the node supervising the corresponding interval.

```
function StoreData(data,interval)
    if $\exists i : interval = u.DS.i$ then
        u.DS[i].data := $u.DS[i].data \cup data.id$
        linearize(u.DS[i].ref)
        u.DS[i].ref=$\mathrm{argmax}_{w.id}\{w \in u.N : w.id < data.id\}$
    else
        l:=$|u.DS|$
        u.DS[l+1]=interval
        u.DS[l+1].ref=$\mathrm{argmax}_{w.id}\{w \in u.N : w.id < data.id\}$
        u.DS[l+1].data=data
```

5 Correctness

In this section we show the correctness of the presented algorithm. 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.

5.1 Convergence

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. For our analysis we additionally define $E(t)$ as the set of edges at time t. Analogous $E_e(t)$ and $E_i(t)$ are defined. We show the following theorem.

Theorem 5.1 *After $\mathcal{O}(\log^4 n)$ steps, the network has reached a state, where the explicit network graph is a super-graph of G_{CONE}.*

We divide the proof into 2 phases. First we show the convergence to the sorted list and eventually the convergence to the *CONE*-network.

Phase 1: Towards the Sorted List

For the rest of the analysis we assume that all variables of each node are valid according to Definition 3.1, i.e. we assume that each node has performed one $validitycheck()$ and all nodes maintain valid neighborhoods. We consider this time step, where each node maintains a valid neighborhood as t_0. In this phase we show that eventually all nodes form a sorted list as a sub-topology. We therefore define

$G^{List} = (V, E^{List})$, where $E^{List} = \{(u,v) : (u,v) \in E_e \wedge (v = \text{argmin}\{w.id : w.id > u.id\} \vee v = \text{argmax}\{w.id : w.id < u.id\})\}$.

So, we need to prove the following theorem.

Theorem 5.2 *After $\mathcal{O}(\log^4 n)$ steps, the network has reached a state, where the explicit network graph is a super-graph of G_{List}.*

In order to do this, we show that the following lemma holds.

Lemma 5.3 *Let a time step $t \geq t_1 = 82\log n + t_0$ and a node u being at a linear path $\in lp_t(a,b)$. Then at time $t_j = t_1 + i \cdot h, i > 0$ (for a sufficiently large constant h) it holds w.h.p. that u, as well as its right (resp. left) links (if it has any of them) of order $\leq i$ such that $1/2^{\lceil log(n) \rceil - i + 1}$ (i.e. the length of the i-th order right, resp. left link) is smaller than $\frac{b.id - u.id}{2}$ (resp. $\frac{u.id - a.id}{2}$), are on a linear path between a and b.*

Proof.

We prove the lemma by induction over i.

Induction basis :

We start the induction at round $t_1 = 82\log n + t_0$. As we have shown, after $82\log n$ rounds all the nodes will have already had a complete probing procedure and will have reset their probing procedure, and at least one new has been started. W.l.o.g. we restrict our analysis only to the case of the right links. If u already has a right link of order 1 at a linear path $p \in lp_{t_1}(a, b)$ we are done. So let's assume this is not the case.

Let v be the closest node to u from the right of p. In case u does not have an edge to v (then obviously v has an edge to u) we consider following subcases. If (v, u) is an explicit edge, then v will contact u during its next probing procedure, which happens in $O(\log n)$ rounds w.h.p. If (v, u) is an implicit edge, then following cases are possible.

- The edge can be a (u, lin) message, in which case v sends u a lin message with v's closest neighbor from the left.
- This edge can be a $(u, probingr)$ message, in which case v would linearize u or contact u during the next probing otherwise if u is the first argument. If u is the $m.sender$ argument, it is handled analogously.
- The implicit edge (v, u) could be a $requestprobing$ message then a $continueprobing$ message is sent back to u.
- The edge can otherwise be a $(u, rightring)$ message, in which case it means that u sent that message and has a ring edge to v. In case the ring edge was dropped it would have been linearized before by u.
- The edge could be an $(u, requestneib)$ message, in which case v sends $v.N$ to u.
- The edge could be a $(u, checkinterval)$ message, in which case v would call $linearize(u)$, which is identical to the previous case of v getting a (u, lin) message.
- The edge could be a $update - list(List)$ message, with $u \in List$. Then if v has not already saved u in one of its lists, a lin message is sent to u. In case v is in one of u's lists, a (u, v) edge exists anyway.

In any case, after $O(\log n)$ time steps there exists an edge (u, v).

Now, either is v the first direct right neighbor of u (the leftmost of $u.rDirect$) or u maintains another node which is that, let that be w. Either during the last probing procedure, or when u learns about v, v is introduced to w, which means that after a constant number of rounds the first direct right neighbor of u is at a linear path from a to b. By continuing this argumentation analogously, we conclude that all nodes in $u.rDirect$, as well as all nodes in $u.P$ are between u and $rlink_1(u)$ (including $rlink_1(u)$) are at a linear path from a to b after $O(\log n)$ rounds (since there lie $O(\log n)$ nodes between u and $rlink_1(u)$ w.h.p.) w.h.p.

Induction step:

According to the induction hypothesis, at time $t_1 + (i-1)h$, u has already links of order $< i$ at some $p_x \in lp_{t_1+(i-1)h}(a, b)$.

If the link $rlink_i(u)$ is already on the path or $1/2^{\lceil log(n) \rceil - i + 1} > \frac{b.id - u.id}{2}$, then we are done. Else, we know from the induction hypothesis that u maintains $rlink_{i-1}(u)$ at some $p_y \in lp_{t_1+i-1}(a, b)$, if u was already on such a path. Also, according to the induction hypothesis, $rlink_{i-1}(u)$ maintains also its link $rlink_{i-1}(rlink_{i-1}(u))$ on some path $\in lp_{t_1+i-1}(a, b)$ (see Figure 3.2), if $1/2^{\lceil log(n) \rceil - i} < \frac{b.id - rlink_{i-1}(u)}{2}$. (If this is not the case then also $1/2^{\lceil log(n) \rceil - i + 1} > \frac{b.id - u.id}{2}$, and then we are done.)

Now, one case is if $rlink_{i-1}(rlink_{i-1}(u))$ and u have been introduced to each other by $rlink_{i-1}(u)$, at timestep $t_j > t_0$ when $rlink_{i-1}(u)$ learned about u (or $rlink_{i-1}(rlink_{i-1}(u))$), according to the $(rlink_{i-1}(u)).linearize(u)$ action (or $(rlink_{i-1}(u)).linearize(rlink_{i-1}(rlink_{i-1}(u)))$ action). Then the two nodes are introduced to each other because $(rlink_{i-1}(u)).N$ would be unequal to $(rlink_{i-1}(u)).Nold$. In case $rlink_{i-1}(rlink_{i-1}(u))$ and u have been neighbors of $rlink_{i-1}(u)$ already at t_0, then at most at timestep $t_1 = 4log(n) + t_0$ during the probing procedure of u node $rlink_{i-1}(u)$ will have been reached and $rlink_{i-1}(rlink_{i-1}(u))$ will have been introduced to u according to the action $(rlink_{i-1}(u)).continueprobingright$ or $(rlink_{i-1}(u)).continueprobingleft$, where the closest node to the probing goal is sent to u, which in this case is $rlink_{i-1}(rlink_{i-1}(u))$. Since $(u, rlink_{i-1}(u))$ and $(rlink_{i-1}(u), rlink_{i-1}\ (rlink_{i-1}(u)))$ have at least $1/2^{log(n)-(i-1)+1}$ length each, then the new edge $(u, rlink_{i-1}(rlink_{i-1}\ (u)))$ will have length at least $1/2^{log(n)-i+1}$, which means that it is stored as $rlink_i(u)$, if u does not already have one. If u did already have a $rlink_i(u)$ l at t_0, then the following cases can occur. In case $l.id > rlink_{i-1}(rlink_{i-1}(u)).id$ then l is dropped as a link and we are done. Otherwise, $rlink_{i-1}(rlink_{i-1}(u))$ is forwarded to l and again the lemma holds. Note that the correctness of the lemma is not influenced if during this time $rlink_{i-1}(u)$ (or analogously/as a consequence $rlink_{i-1}(rlink_{i-1}(u))$) changes its value. We will argue here why. Suppose $rlink_{i-1}(u) = v$ until some time step t, and at the same time step u learns a new node w which qualifies better for $rlink_{i-1}(u)$ and sets $rlink_{i-1}(u) = w$. That means, according to the $linearize$ action, that u sends a $request(w)$ message to v and v sends $v.N$ to w at time step $t + 1$. At time step $t + 2$ w receives $v.N$ and either sets $rlink_{i-1}(w) = rlink_{i-1}(v)$ or forwards $rlink_{i-1}(v)$ (i.e. what was $rlink_{i-1}(rlink_{i-1}(u))$) to $rlink_{i-1}(w)$. That happens for all existing right links of w, so all existing right links with id less than $b.id$ are on a path from a to b at time $t+2$. Note that the statement of the lemma for u's link $rlink_i(u)$ is also not influenced, since u would still have its $rlink_i(u)$ link on a linear path from a to b.

That is because in order for the value of $rlink_i(u)$ to be changed to $rlink_{i-1}(w)$ either $rlink_{i-1}(w)$ was introduced to u before $t+2$, or u learned about $rlink_{i-1}(w)$ from w during the probing process. But in the latter case this cannot happen until $t+2$. That is because at time step t u learns about w, so it can at earliest send a continueprobing message to w, which will receive this message at $t+1$ and respond at $t + 2$ by sending (possibly) $rlink_{i-1}(w)$ to u. Now, let's consider the other case, where $rlink_{i-1}(w)$ was introduced to u before $t + 2$. Here, we consider 2 subcases. In the first sub-case u has already had another $rlink_i(u)$ when $rlink_{i-1}(w)$ was introduced, so it just changed the value of $rlink_i(u)$ to $rlink_{i-1}(w)$ and forwarded $rlink_{i-1}(w)$ to the previous $rlink_i(u)$, so the lemma still holds. The second sub-case is that u's $rlink_i(u)$ was null when it learned about $rlink_{i-1}(w)$. But that means that $t + 1 < t_1 + i \cdot h$, so the lemma holds.

□

Further we show following lemma, which concludes the proof of the theorem.

Lemma 5.4 *At some time step $t_2 = t_1 + c\log^3 n$, for some constant c, the sorted list G^{List} has been formed as a sub-graph of the network graph G_{t_2}.*

Proof. Same as proof of Lemma 3.7. □

Phase 2: From the Sorted List to the Desired Topology

In this section we show that once the network has stabilized into a sorted list, it eventually also stabilizes into a legal Cone-network state, that means, each node u maintains a correct set of neighbors, so the lists $u.P^+, u.S^+, u.P^-, u.S^-$ as well as the fingers maintain the correct nodes, so for example the list $u.P^+$ maintains the nodes in $P^+(u)$.

Due to Lemma 5.3 we know that at most $\mathcal{O}(\log n)$ time steps after t_2, each node u on the sorted list maintains its right (resp. left) fingers of order i such that $u.id + 1/2^{\lceil log(n)\rceil - i+1} < b.id$, with b being the rightmost node in the list, i.e. with the maximal id (resp. $u.id - 1/2^{\lceil log(n)\rceil - i+1} > a.id$, with a being the leftmost node, i.e. with the minimal id).

Lemma 5.5 *The ring is formed as a sub-graph of the network graph at time step $t_2 + d\log n$, for some constant d.*

Proof. Same as proof of Lemma 3.8. □

Lemma 5.6 *At time step $t_3 = t_2 + c'\log n$ a node u is connected to its correct right and left $u.mylogn$ direct neighbors, i.e. $u.rDirect = rDirect(u)$ and $u.lDirect = lDirect(u)$, for a sufficiently large constant c'.*

Proof. Same as proof of Lemma 3.9. □

Lemma 5.7 *At time step $t_4 = t_3 + f\log n \cdot i$ each node u is connected to its correct right (and left) link of order i (i.e. $rlink_i(u) = rfinger_j(u) = u.rfinger_j$, for some $u.first \leq j \leq u.mylogn$), for a sufficiently large constant f.*

Proof.
Same as proof of Lemma 3.10.
□

We proceed with the following definition.

Definition 5.8 *We say that a node u has right (resp. left) level i (rlevel(u)=i (resp. llevel(u)=i)), if it is i steps away from the node on its right with the largest capacity, i.e. $|P^+(u)| = i$ (resp. $|S^+(u)| = i$).*

We now need to show the following lemma.

Lemma 5.9 *At time step $t_2 = t_1 + d\log^4 n$ for some constant d, it holds that every node u knows its correct S^+ and P^+ lists.*

Proof.
Let us first show that at time step $t_1 + d \cdot i \cdot \log^3 n$ for some constant d, it holds for every node u of right and left level $\geq i$ that $u.succ_1^+ = succ_1^+(u)$ and $pred_1^+(u) = u.pred_1^+$.
We will show that statement by induction.

W.l.o.g. we conduct the induction for the case of the right level.
The induction starts for $i_{max} = \max_{\forall w}\{level(w)\}$ and is done backwards (from i to $i-1$).

Induction basis: At t_1, for each node u of right level $i_{max} = \max_{\forall w}\{level(w)\}$ it must hold that u and $succ_1^+(u)$ are direct neighbors on the sorted list. That is because there cannot be another node between u and $succ_1^+(u)$ with less capacity than u, otherwise u would not be of right level

$i_{max} = \max_{\forall w}\{level(w)\}$. Also, there cannot be another node between u and $succ_1^+(u)$ with larger capacity than u, otherwise that node would be $succ_1^+(u)$.

Inductive step: We argue that if at some time step every node with right level i knows $u.succ_1^+$, then after $\mathcal{O}(\log^3 n)$ steps each node u with right level $i-1$ knows also $succ_1^+(u)$. That is due to the following:

Note that $succ_1^+(u).P^-[1] = P^-[1](succ_1^+(u))$, since it is its direct neighbor. We will further argue that $P^-[1](succ_1^+(u)) = x$ and $P^-[2](succ_1^+(u)) = y$ are connected at that time point. let's take a look at $y.S^-$. Note that all nodes in $y.S^-$ are of right level $\geq i$, so they are connected with their capacity successors, according to the induction hypothesis. The first one (from the left) of these nodes is $y.S^-[1]$, which is connected to y (as they are direct neighbors) and to $y.S^-[2]$ (according to the induction hypothesis). At most after $\mathcal{O}(\log n)$ time steps $y.S^-[1]$ will introduce y to $y.S^-[2]$ during its $(y.S^-[1]).BuildTriangle()$ call and $y.S^-[2]$ will set y as $(y.S^-[2]).pred_1^+$. Analogously, after $\mathcal{O}(\log n)$ rounds $y.S^-[3]$ will set y as $(y.S^-[3]).pred_1^+$ and within $\mathcal{O}(\log^2 n)$ rounds x will set also y as $x.pred_1^+$ (since $|y.S^-| = O(\log n)$ w.h.p.) and so x will introduce $x.succ_1^+$ (which also is $succ_1^+(u)$) to y during its periodic $x.BuildTriangle()$ call, at most after $\mathcal{O}(\log n)$ time steps.

Analogously $P^-[2](y = succ_1^+(u))$ will introduce $succ_1^+(u)$ to $P^-[3](succ_1^+(u))$ within the next $\mathcal{O}(\log^2 n)$ steps, and so on, until after $\mathcal{O}(\log^3 n)$ steps (since $|succ_1^+(u).P^-| = O(\log n)$ w.h.p.) the last node in $u.S^-$, let's call it z, will also be introduced to $succ_1^+(u)$ (since it is also part of $P^-(succ_1^+(u))$). So z will set $z.succ_1^+ = succ_1^+(u)$. In its periodic $z.BuildTriangle()$ action, after at most $\mathcal{O}(\log n)$ time steps, z will introduce $z.succ_1^+ = succ_1^+(u)$ to u and the induction statement holds.

Now, since we have seen that the number of levels is $\mathcal{O}(\log n)$ w.h.p., it holds that after $\mathcal{O}(\log^4 n)$ after t_1, every node u knows its correct right and left direct successor/predecessor ($succ_1^+(u)/pred_1^+(u)$).

Now, once each node u maintains its correct right and left direct successor/predecessor ($succ_1^+(u)/pred_1^+(u)$), it periodically (at most every $\mathcal{O}(\log n)$ time steps during the periodic action) receives a $list - update$ message from them. So the correct right and left lists are propagated from the top capacity nodes to the low capacity nodes every $\mathcal{O}(\log n)$ time steps, which means that after $\mathcal{O}(\log^2 n)$ (since the length of the P^- and S^- sets are also $\mathcal{O}(\log n)$ w.h.p.) time steps all nodes have been informed about their correct S^+ and P^+ lists.

And so the lemma holds. □

Lemma 5.10 *If every node knows its correct closest larger right (resp. left) node $succ_1^+(u)$ (resp. $pred_1^+(u)$) stored in $u.succ_1^+$ (resp. stored in $u.pred_1^+$) then for all nodes x which are in the correct right (resp. left) internal neighborhood of u, $S^-(u)$ (resp. $P^-(u)$), it holds that u will learn x (and store it in $u.S^-$ (resp. $u.P^-$) after $\mathcal{O}(\log^2 n)$ time steps w.h.p..*

Proof. W.l.o.g. we consider only the proof for $u.S^-$.
We will prove it by induction over the nodes $x \in S^-(u)$ (in ascending order of their $x.id$ values).

Induction basis: x is the direct right neighbor of u. In that case u already knows x, since we assumed the presence of the sorted list, and the statement holds.

Inductive step: We need to show that if u knows the next node $\in S^-(u)$ to the left of x (let this be y), then u learns x after $\mathcal{O}(\log n)$ time steps.

In this case, x is the closest larger right node of y. That is because x must be larger (in terms of capacity) than y, since else x would not be in $S^-(u)$. So, by hypothesis, y knows about x (so

$y.succ_1^+ = x$). So, when y conducts its periodic $y.BuildTriangle()$ call after $\mathcal{O}(\log n)$ time steps, it will introduce u and x to each other (as they are $y.pred_1^+$ and $y.succ_1^+$ respectively) and u will learn about x.

Now, since the size of $S^-(u)$ is $\mathcal{O}(\log n)$ w.h.p. the lemma holds. $\square$

Combining the previous lemmas,we can show that Theorem 5.1 holds, and by our protocol each weakly connected network converges to the CONE network.

Lemma 5.11 *Redundant explicit edges disappear after $\mathcal{O}(\log n)$ steps.*

Proof. Here we show that explicit edges that do not belong to G_{CONE} disappear over time, so that the set of explicit edges forms the G_{CONE}.

So let's consider the time step t_4 when the graph of the explicit edges G forms a super-set of G_{CONE}. Let an explicit edge $(u, v) \in G$, which is not in G_{CONE}. That means that v does not belong in the correct $N(u)$ in any way and will be forwarded through the $linearize$ action to the closest node of $u.N$, w. Since u maintains all its correct fingers, $distance(w, v) \leq \frac{1}{2}distance(u, v)$. Since this happens continuously the distance of the edge to v is halfed in each step, thus after $\mathcal{O}(\log n)$ steps v will be connected with its direct neighbor and the redundant edge no longer exists. $\square$

5.2 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 5.12 *If $G_e = G^{CONE}$ at time t then for $t' > t$ also $G_e = G^{CONE}$.*

Proof. The actions conducted periodically are the on the one hand $linearize()$ and the probing actions. Since the nodes maintain their correct neighborhoods (due to the fact that no external dynamics occur), the $linearize()$ action does not alter the explicit graph. The only way the probing action can add a new explicit edge is if it fails. However due to Lemma 3.4 this does not happen w.h.p.. $\square$

Theorem 5.13 *If $G_e = G^{CONE}$ then after $\mathcal{O}(\log n)$ time steps each node stores exactly the data it is responsible for.*

Proof. According to Theorem 2.4 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 each $\Theta(\log n)$ time steps 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 reaches after $\mathcal{O}(\log n)$ time steps a node supervising the corresponding interval. This node then tells the responsible node to store the data. Thus all data is stored by nodes that are responsible for the data. $\square$

6 Performance Results

6.1 Message Complexity

Theorem 6.1 *The nodes in the network send (and thus receive) $\mathcal{O}(n^2 \log^5 n)$ messages (of logarithmic size) during the self-stabilization.*

Proof. The self-stabilization process takes $\mathcal{O}(\log^4 n)$ steps. In each step each node can send at worst case all of its neighborhood ($\mathcal{O}(\log n)$) to all other nodes in the graph $(n-1)$. Thus the Theorem follows. □

Theorem 6.2 *In the stable state, $\mathcal{O}(d/\log n)$ number of messages are sent/received per node per time step on average, where d is the number of data assigned to a node.*

Proof. In the stable state, first of all messages associated with probing are sent. Obviously a node sends only one $probingl$, $probingr$ message in each time step. Also to each node exactly one $continueprobingright$ and $continueprobingleft$ message is sent in each time step from some other node. Thus each node sends and receives on average a constant number of messages per time step.

Moreover, in the stable state, each node u sends two $(List, list-update)$ messages every $\Theta(\log n)$ number of rounds. However in the stable state a $list-update$ message received does not cause any further messages to be sent, since the maintained lists are correct. The argument $List$ in a $list-update$ message is of size $\mathcal{O}(\log n)$ (since this is the size of $u.P^+, u.S^+$) w.h.p., so at average, the communication complexity caused by the $(List, list-update)$ messages is $\mathcal{O}(1)$ messages (of logarithmic size) per node per time step.

Moreover, in the stable state, u calls the $BuildTriangle()$ function every $\Theta(\log n)$ number of rounds. This causes to sending a (u, lin) message to all nodes in $u.S^- \cup u.P^- \cup \{u.succ_1^+, u.pred_1^+\}$. However, since all these nodes also maintain u in their neighborhood in the stable state, this does not cause any further messages to be sent. Also, a $(u.pred_1^+, lin)$ message is sent to $u.succ_1^+$ and vice versa. Again, $u.succ_1^+$ and $u.pred_1^+$ maintain each other in their neighborhoods, so this does not cause any further messages to be sent. Since the size of $u.S^-$ and $u.P^-$ is $\mathcal{O}(\log n)$ w.h.p., if follows that the $BuildTriangle()$ function results to $\mathcal{O}(1)$ messages at average per node per time step.

Moreover, in the stable state, u calls the $CheckDataInterval()$ function every $\Theta(\log n)$ number of time steps. Then, a $check-interval$ message is sent to all the intervals u (thinks it) is responsible for. In the stable state, u is indeed responsible for these intervals, so no further messages will be sent. This results to an average of $\mathcal{O}(d/\log n)$ messages sent per node per time step due to the $CheckDataInterval()$ function, and the proof is concluded.

□

7 External Dynamics

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 7.1 *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 $\mathcal{O}(\log^2 n)$ structural changes in the explicit edge set are necessary to reach the new stable state.*

Proof.
We show the statement by considering the 3 cases separately.

7.1 Joining of a Node

When a new node u enters the network, it does so by maintaining a connection to another node v, which is already in the network. u is forwarded due to the periodic actions in the network until it reaches its right position, as it takes part in the linearization procedure.

Theorem 7.2 *If a node u joins a stable CONE network $\mathcal{O}(\log^2 n)$ structural changes in the explicit edge set as well as $\mathcal{O}(\log^2 n)$ time steps are necessary to reach the new stable state.*

Proof. We show that there is at most a constant number of temporary edges, i.e. edges that are not in the stable state. u stores v in $u.N$ as v is the only node u knows. In its periodic action u after $\mathcal{O}(\log n)$ time steps (either through probing, through a $list - update$ message or through the $BuildTriangle()$ call) sends a message to v containing its own id creating an implicit edge (v, u). Now there can be two cases: Either u is in $v.N$ in a stable state then v stores u's identifier or u is not stored and delegated to another node w creating the implicit edge (w, u). Thus only explicit edges pointing to u are created that are in the stable state and only the explicit edge (u, v) is temporary. Each time u is delegated the distance to its direct neighbors is at least halfed, due to the routing structure (use of the fingers) of the network. Thus after $\mathcal{O}(\log n)$ time steps u will have reached its directs neighbors. Then u will periodically conduct probing and maintain its fingers. For each finger to reach its correct position at most one probing is sufficient (again due to the distance-halfing network structure). So after $\mathcal{O}(\log^2 n)$ time steps u will maintain its correct fingers. Due to the $BuildTriangle()$ call conducted by the nodes in $S^-(u)$ and $P^-(u)$ u learns every $\mathcal{O}(\log n)$ time steps at least one of these nodes, so after $\mathcal{O}(\log^2 n)$ time steps u maintains its correct $u.S^-$ and $u.P^-$ lists as well as its correct $u.succ_1^+$ and $u.pred_1^+$. Through the periodic $list - update$ messages of $u.succ_1^+$ and $u.pred_1^+$ u also learns its correct $u.S^+$ and $u.P^+$ lists. Every $\Theta(\log n)$ time steps u will check the correctness of its data intervals and in case they are wrong it receives the correct ones.

Note here that after the joining of u in the network also edges that have been before in the stable state no longer exist in the new stable state. E.g. let $w \in x.S^+$ and $x.id < u.id < w.id$ and $c(u) > c(w)$ then w is not longer stored in $x.S^+$ as soon as u is integrated in the network. According to Theorem 2.5 there is at most $\mathcal{O}(\log n)$ w.h.p. such nodes x, as each node x has to store u in its lists, and also at most $\mathcal{O}(\log n)$ w.h.p. nodes w, as u has to store each w in its lists. Therefore there are at most $\mathcal{O}(\log^2 n)$ edges that have to be deleted. The nodes which have lists that are affected from u's arrival are the ones being in $u.S^-$ and $u.P^-$, which will learn u after $\mathcal{O}(\log^2 n)$ time steps and get their corresponding correct lists from u at most $\mathcal{O}(\log n)$ time steps after that. So all in all we showed that $\mathcal{O}(\log^2 n)$ structural changes in the explicit edge set as well as $\mathcal{O}(\log^2 n)$ time steps are necessary to reach the new stable state. □

7.2 Node Departures

In contrast to the previous chapter, in this chapter we cannot directly apply the algorithmic framework of chapter 2 for solving the $\mathcal{FDP}$ because the protocol does not fulfill the requirements. However we will show that despite that the framework still works.

Theorem 7.3 *If the algorithmic framework for solving $\mathcal{FDP}$ of chapter 2 is applied to to the protocol presented in this chapter, then as a result we get a protocol that solves $\mathcal{FDP}$.*

Proof. We can overcome the unfulfilled requirement that the protocol does not include self-introduction of a node to all of its neighbors periodically. Note that a node u does introduce itself to all its neighbors, but not to the ones in $u.S^+$ and $u.P^+$. W.l.o.g. we consider $u.S^+$. We will show that if there is a node v in $u.S^+$ which is leaving, then u is been eventually informed about that and dissolves the edge to v.

u gets periodically informed by $u.succ_1^+$ about $(u.succ_1^+).S^+$. Now there are two cases. Either $u.succ_1^+ = v$ or not. In the first case we know that eventually u will contact v and so v will respond that it is leaving, which means that u will give up v and reverse its edge to it.

Now in case that $u.succ_1^+ \neq v$ we can distinguish between two subcases. Either v is included in $(u.succ_1^+).S^+$ or not. If v is not included, then u updates $u.S^+$ (not including u) and forwards itself to v, which means that u no longer points to v. If on the other hand v is included in S, then we are back to same situation and the arguments for u above apply to $u.succ_1^+$. However this situation can occur only a finite number of times, since the number of nodes is finite. □

Note that in addition to the framework we also include an extra action that forwards the data stored in u to one of its neighbors, before a node u leaves.

After the leaving, the network must stabilize again. This means that $u.S^-[1]$ and $u.P^-[1]$ must connect to each other. let's consider $u.P^-[1]$. Since it won't have a direct right neighbor after the leaving of u, the linearization process will take place again until $u.P^-[1]$ learns $u.S^-[1]$.

Theorem 7.4 *If a node u leaves a stable CONE network $\mathcal{O}(\log^2 n)$ structural changes in the explicit edge set as well as $\mathcal{O}(\log^2 n)$ time steps are necessary to reach the new stable state.*

Proof. The proof is analogous to the proof in the case of a joining node. Obviously according to Theorem 2.5 w.h.p. $\mathcal{O}(\log n)$ edges are deleted that start at or point to the leaving node u. By deleting u further edges have to be created. E.g. let $w \in u.S^-$ and $u \in x.S^+$ and $c(u) > c(w)$ then w might now be stored in $x.S^+$ or $x.S^-$ and the edge (x, w) has to be created. Again according to Theorem 2.5 there are w.h.p. at most $\mathcal{O}(\log n)$ such nodes x and $\mathcal{O}(\log n)$ such nodes w, thus in total at most $\mathcal{O}(\log^2 n)$ edges have to be created.

Analogously to the previous proof, due to the $BuildTriangle()$ and $list{-}update$ calls conducted by these nodes, each of them needs $\mathcal{O}(\log n)$ structural changes in order to maintain its correct lists as well as $\mathcal{O}(\log^2 n)$ time steps.

Moreover, nodes that had u as a finger or direct neighbors must renew their neighborhood. These are $\mathcal{O}(\log n)$ w.h.p., according to Lemma 2.10. Let a node y which had u as a finger. After $\mathcal{O}(\log n)$ time steps the probing will find a new finger, which is will be one of the former direct neighbors of u. So after each probing at least one correct finger is found by a node, which means that at most after $\mathcal{O}(\log^2 n)$ time steps all correct fingers are found. □

7.3 Capacity Change

If the capacity of a single node u in a stable CONE network decreases we can apply the same arguments as for the leaving of a node, as some nodes might now be responsible for intervals that u was responsible for. Additionally u might have to delete some ids in $u.S^- \cup u.P^-$ and add ids in $u.S^+ \cup u.P^+$. If a node increases its capacity we can apply the same arguments as for the joining of a node, as some nodes might not longer be responsible for intervals that u is now responsible for. Additionally u might have to add some ids in $u.S^- \cup u.P^-$ and delete ids in $u.S^+ \cup u.P^+$. Thus the following theorem follows.

Theorem 7.5 *If a node u in stable CONE network changes its capacity $\mathcal{O}(\log^2 n)$ structural changes in the explicit edge set as well as $\mathcal{O}(\log^2 n)$ time steps are necessary to reach the new stable state.*

□

8 Outlook

In this chapter we considered heterogeneity in a DHT, utilized as a distributed storage system. More specifically, the heterogeneity concerned the storage capacity of the nodes. We gave a self-stabilizing algorithm for a heterogeneous overlay network, and by doing this we used an efficient network structure. The network fairly distributes the data load to the nodes according to their storage capacity. The algorithm is based on the one presented in $DISC$ 2013, but improves it by removing the previous unequally distributed load balance due to routing. Improvement is also done in the self-stabilization time, since the new algorithm only needs a polylogarithmic number of steps. We proved the correctness of our protocol, also concerning the functionality of the operations done in the network, data operations and node operations. This was 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 low degree, fair load balancing, and polylogarithmic update cost in case of joining or leaving nodes. More work in this field can be done by examining heterogeneous networks in the two-dimensional space and considering heterogeneity in other aspects than only the capacity, e.g. bandwidth, reliability or heterogeneity of the data elements.

Chapter 5

The Self-Stabilizing Resource Discovery Problem

A Deterministic Worst-Case Message Complexity Optimal Solution

Until now we considered specific network topologies, as goal topologies of the self-stabilizing process. In this chapter we consider a pretty straight forward topology as our goal topology, the clique, or as it is called in the field of distributed systems, the problem of resource discovery. In particular we give an algorithm, such that each node in a network discovers the address of any other node in the network. Although there are several solutions for resource discovery, our solution is the first that achieves worst-case optimal work for each node, i.e. the number of addresses ($\mathcal{O}(n)$) or bits ($\mathcal{O}(n \log n)$) a node receives or sends coincides with the lower bound, while ensuring only a linear run time ($\mathcal{O}(n)$) on the number of rounds.

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 [9], verifiable secret sharing [6], distributed consensus [10], and broadcasting [11] 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 [16].

1.1 Resource Discovery

As mentioned in [16] 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 $\mathcal{O}(\log n)$ 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. We measure the work of a node as the number of addresses each node receives or sends while executing the algorithm. 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 [16] 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 [17] a deterministic solution for resource discovery in distributed networks was proposed by Kutten et al. Their solution uses the same model as in [16] and improves the number of communication rounds 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 $\mathcal{O}(n^2 \log^2 n)$. 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 [13], 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 [16, 17] and leads to an increased run time in the number of rounds. We present a deterministic solution that follows the idea of [13] 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 id, one can think of resource discovery as building and maintaining a complete graph, a clique, as a virtual overlay network. If the overlay can be recovered out of any (weakly connected) initial graph, the corresponding algorithm can be considered to be a *self-stabilizing* algorithm. More precisely, an algorithm is considered self-stabilizing if it reaches a legal state when started in an arbitrary initial state (*convergence*) and stays in a legal state when started in a legal state (*closure*).

1.2 Related Work

In [28] the authors use a self-stabilizing algorithm in which they collect snapshots of the network along a spanning tree, which could also be used to form a complete graph. However, the authors give no bounds on the message complexity of their algorithm. In [3] 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 and may involve the construction of a clique. 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 too high as they essentially use pointer doubling: i.e. in each round a node sends the information about its neighborhood to all its neighbors.

One could use the distributed algorithms for self-stabilizing lists and rings to form a complete graph, but all algorithms proposed so far for these topologies involve a worst-case work of $\Omega(n^2)$ per node in order to form the list or ring. Hence, these algorithms cannot be used to obtain an efficient algorithm for the clique.

Alternatively, a self-stabilizing spanning tree algorithm could be used. A large number of self-stabilizing distributed algorithms has already been proposed for the formation of spanning trees in static network topologies, [5], [4], [8]. For example, in [5] the authors present a self-stabilizing spanning tree with minimal degree for the given network and in [4] a fast algorithm for a self-stabilizing spanning tree is presented, which reaches optimal convergence time $\mathcal{O}(n^2)$ in an asynchronous setting. However, these spanning trees are either expensive to maintain or the amount of work in these algorithms is not being considered.

However, these spanning trees are potentially expensive to maintain as a high degree cannot be avoided in general (consider, for example, the extreme case of a star graph in which a single node is connected to all other nodes). 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., [8]). The algorithm in [8] 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.

In summary, no self-stabilizing algorithm has been presented for the formation of a bounded degree spanning tree if the network topology is under the control of the nodes and there are no outside services for the introduction of nodes.

1.3 Our Contributions

In this chapter we present a distributed algorithm for resource discovery. We will describe the algorithm as a self-stabilizing algorithm that forms and maintains a clique as a virtual overlay network, out of any weakly-connected initial state. In particular, we show that our algorithm is worst-case optimal in terms of message complexity. More specifically we show that for any initial state in which the network is weakly connected, our algorithm requires at most $\mathcal{O}(n)$ rounds and $\mathcal{O}(n)$ work per node until the network reaches a legal state in which it forms a clique.

We further show that the maintenance cost per round is $\mathcal{O}(1)$ 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. Note that we use a synchronous message passing model to give bounds on the message complexity of our algorithm, but our correctness analysis can also be applied to an asynchronous setting.

2 A Distributed Self-Stabilizing Algorithm for the Clique

In this section we give a general description of our algorithm. First we introduce the variables being used, and then the actions the nodes take, according to our rules. To each node x corresponds a channel $x.C$ of incoming messages from the previous round. We do not require any particular order in which the messages are processed in $B(x)$. Moreover, each node x stores the following internal variables: its predecessor $x.p$, its successor $x.s$, its current neighborhood $x.N$ in a circular list, the nodes received by messages from the predecessor in another circular list $x.L$, the set of nodes $x.S$ that are received through scanning messages (defined below), its own identifier $x.id$ and its status $x.status$, which is by default set to 'inactive' and can be changed to 'active'. The current network $G = (V, E)$ formed by the nodes is defined by their current neighborhoods $v.N$. We only require that $v.N$ does not contain false ids. By that we mean ids of nodes that do not exist. If we would allow this, stabilization time could be delayed, since there would be additional ids which would have to be propagated by the algorithm.

A message in general consists of the following parts: a *sender id*, which is the id of the node sending the message, an optional *additional id*, if the sender wants to inform the receiving node about another node, and the *type* of the message.

Each node has two different kinds of actions that we call *receive* actions and *periodic* actions. A receive action is enabled if there is an incoming message of the corresponding type in $x.C$. There are the following types of messages: *pred-request, pred-accept, new-predecessor, deactivate, activate, forward-from-successor, forward-from-predecessor, forward-head, scan, scanack, delete-successor*. A periodic action is enabled in every state, as its guard is simply *true*. Therefore there can be no state in the computation in which no action is enabled. Each enabled action is executed once every step.

We can now formally state our main result as following:

Theorem 2.1 *For any initial state in which the network is weakly connected, our algorithm requires at most $\mathcal{O}(n)$ rounds and $\mathcal{O}(n)$ work per node until the network reaches a legal state, in which for all x in the system, $x.N$ is equal to the set of all nodes in the system.*

2.1 Definitions

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 larger identifier in its neighborhood. Therefore for all $x.p$ links, $(x.p).id > x.id$. Then all nodes in a connected component considering only $x.p$ links form a rooted tree, where for each tree the root has the largest identifier. Note here that the *heap H* (defined below) is not a data structure or variable stored by any node. It is a notion used just for the purpose of the analysis.

Definition 2.2 *We call such a rooted tree formed by $x.p$ links a* heap H. *We further call the root of the tree the* head h *of the heap H. We further denote with $heap(x)$ the heap H such that $x \in H$.*

Definition 2.3 *A* sorted list *is a heap H with head h, such that $\forall v \in H - \{h\} : (v.p).id > v.id$ and $\forall v \in H - \{h\} : (v.p).s = v$. We call a heap* linearized w.r.t. *a node $u \in H$, if $\forall v \in H - \{h\} : (v.p).id > v.id$ and $\forall v \in H - \{h\} \wedge v \geq u : (v.p).s = v$. 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.*

2.2 Description of our Algorithm

We only present the intuition behind our algorithm. The full pseudocode is in Appendix 2.3. Our primary goal is to collect the addresses of all nodes in the system at the node of maximum id, which we also call the *root*. In order to efficiently distribute the addresses from this root to all other nodes in the system (so that all ids are known to every node and a clique is formed), we aim at organizing them into a spanning tree of constant degree, which in our case is a sorted list, ordered in descending ids. The root would then be the head of the list. In order to reach a sorted list, we first organize the nodes in rooted trees satisfying the max-heap property, i.e. a parent (also called *predecessor* in the following) of a node has a higher id than the node itself. The rooted trees will then be merged and linearized over time so that they ultimately form a single sorted list.

Since we want to minimize our message complexity, we had to look for a technique other than the linearization technique presented in [35]. So in our protocol, in order to minimize the amount of messages sent by the nodes, we allow a node in each round to share information only with its immediate *successor* $x.s$ (which is one of the nodes that considers it as its predecessor) and *predecessor* $x.p$. More precisely, in each round a node forwards one of its neighbors (i.e. the nodes it knows about) in a round-robin manner to its predecessor. The intuition behind this is that if every node does that sufficiently often, eventually the root will learn about all ids in the system and will forward this information in a round-robin manner to its successor, who will then forward it to its successor, and so on.

In order for this process to work, each node must repeatedly compute and update its successor and predecessor. This is done as follows: Each node chooses the smallest node in its neighborhood that is larger than itself as its predecessor and requests from it to accept it as successor ($pred - request$ message). Each node also looks at the nodes which requested to be its successor, assigns the largest of them as its successor ($pred - accept$) and forwards the rest to it ($new - predecessor$). In that way each node has at most one predecessor and one successor at the end of one round.

We also need to ensure that there exists a path of successors from the root to all other nodes so that the information can be forwarded to all. This is initially not the case since there exist many nodes that are the largest in their known neighborhood, thinking they are the root. We call these nodes $heads$. All the nodes having the same head as an ancestor form a $heap$. The challenge is to $merge$ all heaps into one, since then we have only one head, the root. In order to enable the merging of the heaps, the heads continuously scan their neighborhood. A node that receives a $scan$ message responds by sending the largest node in its neighborhood through a $scanack$ message to the node that sent that $scan$ message (could be possibly more than one). Moreover, in each round, the largest node is also forwarded to its predecessor ($forward - head$), which in turn forwards it again to its predecessor, and so on.

We further discuss the process of forwarding an id to a node's predecessor/successor. Note that when a node forwards an id through a $forward - from - successor$ resp. $forward - from - predecessor$ message, the id sent is the one at the head of the list $x.N$ resp. $x.L$. Then the head shifts to the next element of the (circular) list. When a node receives an id through a $forward - from - successor$ resp. $forward - from - predecessor$ message, it stores it at the head of its list. That way we ensure that once a node is forwarded it will not be delayed by other nodes being forwarded on its way to the root or the head of the heap. When a node is inserted into a list, the $insert$ operation is used. The $insert(< list >, < node >, < place >)$ operation works as follows. It checks whether $< node >$ is already in $< list >$ and if not, it is inserted at $< place >$, where $< place >$ can be either head or tail (by head here the head of the list is meant, not the head of a heap as defined above).

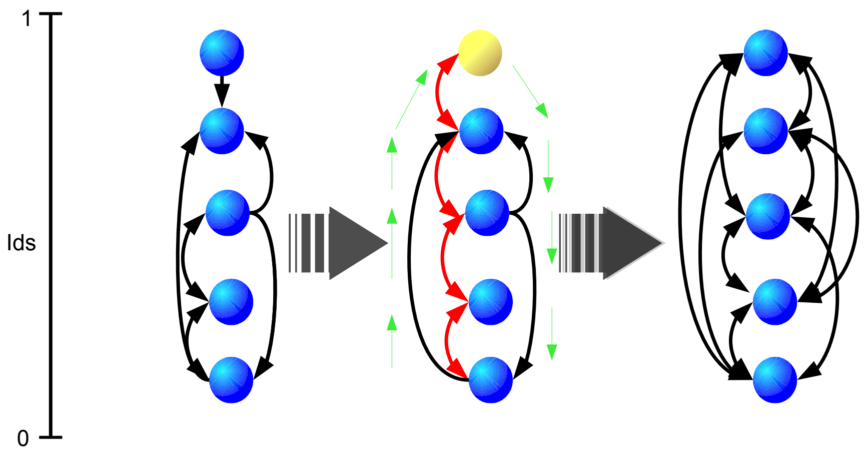


Figure 5.1: Here we depict the nodes by height according to their ids, i.e. nodes with larger id are depicted higher than nodes with smaller id. Our first goal is to form the sorted list as a sub-graph of the network graph (middle part of the picture; the root node is highlighted). Once this has been achieved, the forwarding of the ids through the list (due to the *forward-from-successor* and *forward-from-predecessor* messages) will lead to the construction of the clique (right part of the picture).

To avoid accumulation of unsent ids in the lists (which would have an effect on the time and message complexity) maintained by the nodes, the following rules are used. When x has no predecessor that it can send a *forward-from-successor* message to, although it has neighbors greater than itself (so x is not a head), it changes its status to $inactive$, and then informs its successor through a *deactivate* message in order for $x.s$ not to send its *forward-from-successor* to x, until x has a predecessor (in that case an *active* message is sent to $x.s$) to which it can forward the message. $x.s$ then changes its status to $inactive$ and forwards the *deactivate* message to its successor $(x.s).s$, and so on. In that way no messages that are forwarded to x accumulate at $x.N$ before being forwarded again and we ensure that once a node is forwarded, it will not be delayed by other nodes being forwarded. When x obtains a predecessor, it will change its status to $active$ and inform through a message of type *activate* $x.s$ about that and the information flow can start again.

In order to repair faulty configurations, where a node is thought to be a successor of more than one node, we introduce the following rule. If a node receives messages sent by a node that is not its predecessor although the sending node should be the predecessor, then a node will send a *delete-successor* message, correcting the wrong $x.s$ link.

2.3 Pseudo-Code

In this last section we will present the pseudo code for the described and analyzed algorithm on the next page. The pseudo code starts with the periodic actions (Algorithm 29) and then shows the receive actions (Algorithm 30), in which every incoming message is handled according to the specific message type.

3 Correctness

In this section we show the correctness of our approach for the self-stabilizing clique.

At first we show some basic lemmas. We then show that in linear time all nodes belong to the same heap. Then we show that the head of this heap (node with the maximal id) is connected with every node and vice versa after an additional time of $\mathcal{O}(n)$. From this state it takes $\mathcal{O}(n)$ more time until every node is connected to every other node and the clique is formed. We give a formal definition of the legal state.

Definition 3.1 *Let G be a network with node set V and $max = \max\{v \in V\}$ be the node with the maximum id. Then G is in a* legal state *iff $\forall v \in V : v.N = V - \{v\}$ and $\forall v \in V - \{max\} : (v.p).id > v.id$ and $\forall v \in V - \{max\} : (v.p).s = v$ (i.e. $v.p = p(v)$ and $v.p = s(v)$).*

Note that the legal state contains the clique and also a sorted list over the nodes. In this section we will prove the following theorem.

Theorem 3.2 *After $\mathcal{O}(n)$ rounds the network stabilizes to a legal state.*

3.1 Phase 0: Recovery to a Valid State

In this phase we show that the network can recover if the internal variables $x.p$ and $x.s$ are undefined or set to invalid values, e.g $(x.p).id < x.id$. We therefore define in this case a state as valid state, if the nodes in a connected component given by $x.p$ links form a tree and the successor's predecessor has to be the node itself.

Definition 3.3 *We say that the network G is in a* valid *state if $(x.p).id > x.id$ and $(x.s).id < x.id$ for all $x \in V$ whenever $x.p$ and $x.s$ are defined and if $y = x.s$, then $x = y.p$.*

Theorem 3.4 *It takes at most 2 rounds until the network is in a valid state.*

Proof. The network may be at an invalid state at the first round we consider. That means that the variables $x.p, x.s$ can have invalid values. So x could have set a node u as its predecessor (i.e. $u = x.p$) with $u.id < x.id$, which is not valid according to our protocol. Despite the presence of this invalid state, our protocol can recover from it very fast, so that the actual stabilization procedure can start. So if a variable is set invalid, that is $(x.p).id < x.id$ or $(x.s).id > x.id$, it will be set to $null$ after the first round, once the periodic actions will have been executed, as it is tested in the actions $checkifhead$ and $forwardtosuc$ if $(x.p).id < x.id$ and if $(x.s).id > x.id$. Once each node has computed a valid predecessor, it will request it to accept it as a successor. So after the next round each node will (if possible) also have a valid successor, Moreover if x notices that it is contacted from

Algorithm 29 ACTIONS OF NODE X AT EACH ROUND

```
forwardtopred: true$\rightarrow$
if $x.status \neq inactive \wedge x.p \neq null$ then                              ▷ $x$ is not a head
    send message(x.id, (x.N).head, forward-from-successor) to x.p    ▷ forward node to predecessor
    (x.N).head:=((x.N).head).next                       ▷ shift head to next element in circular list

checkifhead: true$\rightarrow$
if $x.p = null \vee (x.p).id < x.id$ then                          ▷ $x$ is a head or $x.p$ is invalid
    $x.p := argmin_{v \in x.N}\{v.id > x.id\}$
    if $x.p \neq null$ then
        send message(x.id,pred-request) to x.p
        x.status:=inactive
    else                                                        ▷ $x$ is a head, scan a node
        send message(x.id,scan) to (x.N).head
        insert(x.L,(x.N).head,tail)        ▷ a copy of $(x.N).head$ is inserted at the end of $x.L$
        (x.N).head:=((x.N).head).next
else
    send message(x.id,pred-request) to x.p

forwardtosuc: true$\rightarrow$
if $x.s \neq null$ then
    if $(x.s).id < x.id$ then                                         ▷ test if $x.s$ is valid
        send message(x.id, (x.L).head, forward-from-predecessor) to x.s   ▷ forward node to successor
        (x.L).head:=((x.L).head).next
    else
        x.s=null

forwardmax: true$\rightarrow$
if $x.S \neq null$ then
    $maxN := argmax_{u \in x.N}\{u.id\}$
    $x.N := x.N \cup x.S$
    $maxS := argmax_{u \in x.S}\{u.id\}$
    if $maxS > maxN \wedge x.p \neq null$ then                          ▷ forward largest node
        send message(x.id,maxS , forward-head) to x.p
        $S := S \setminus \{maxS\}$
        $maxN = maxS$
    for all $u \in x.S$ do                           ▷ send the maximum to the nodes of $x.S$
        send message(x.id,maxN,scanack) to u
        delete(x.S,u)
```

Algorithm 30 ACTION OF NODE X UPON RECEIVING A MESSAGE

```
process: message $m \in x.C \rightarrow$

if $m.type = forward - head$ then        ▷ insert the head forwarded from $x.s$ to $x.N, x.S$
    if $m.id = x.s$ then
        if $m.id \notin x.N$ then
            insert(x.S,m.id)
        insert(x.N,m.id,tail)

if $m.type = scan$ then        ▷ $x$ has been scanned by a head $m.id$
    insert(x.S,m.id)

if $m.type = scanack$ then
    if $m.id \notin x.N$ then
        insert(x.S,m.id)

if $m.type = delete - successor$ then
    if $m.id = x.s$ then
        $x.s = null$
if $m.type = pred - request$ then
    if $(m.id).id < x.id$ then
        if $x.s \neq null$ then        ▷ renew successor if necessary, and rearrange old successor
            grandson:=min$\{m.id, x.s)\}$
            x.s:=max$\{m.id, x.s\}$
            send message(x,pred-accept) to x.s
            send message(x,x.s,new-predecessor) to grandson
        else
            x.s:=$m.id$
            send message(x,pred-accept) to x.s

if $m.type = new - predecessor$ then        ▷ renew predecessor
    if $m.id = x.p$ then
        if $(m.id2).id > x.id \wedge (m.id2).id < (x.p).id$ then
            x.p=m.id2
            send message(x,pred-request) to x.p
            x.status=inactive
            if $x.s \neq null$ then
                send message(x,deactivate) to x.s

if $m.type = pred - accept$ then        ▷ the predecessor has accepted $x$ as its successor
    if $m.id = x.p$ then
        x.status=active
        if $x.s \neq null$ then
            send message(x,activate) to x.s
    else
        send message(x,delete-successor) to m.id

if $m.type = deactivate$ then
    if $m.id = x.p$ then
        x.status:=inactive
        if $x.s \neq null$ then
            send message(x,deactivate) to x.s        ▷ forward the deactivation message to successor
    else
        send message(x,delete-successor) to m.id
```

Algorithm 31 ACTION OF NODE X UPON RECEIVING A MESSAGE (CONTINUED)

```
if $m.type = activate$ then
    if $m.id = x.p$ then
        x.status:=active
        if $x.s \neq null$ then
            send message(x,activate) to x.s        ▷ forward the activation message to successor
    else
        send message(x,delete-successor) to m.id

if $m.type = forward - from - successor$ then        ▷ insert the node forwarded from $x.s$ to $x.N$
    if $m.id = x.s$ then
        insert(x.N, m.id2, head)

if $m.type = forward - from - predecessor$ then        ▷ insert the node forwarded from $x.p$ to $x.N, x.L$
    if $m.id = x.p$ then
        insert(x.N, m.id2, tail)
        insert(x.L, m.id2, head)
    else
        send message(x,delete-successor) to m.id
```

multiple nodes that think that x has stored them as successors, x contacts all these nodes but one (its true successor) through *delete-successor* messages so at next round x has no multiple successors. In other words it always holds that if $y = x.s$, then $x = y.p$. □

For our further analysis we assume that the initial state is valid, since we do not take into account the first 2 rounds it takes to reach a valid state. So we consider the first round in which we have a valid state as the round $t = 0$. Note that due to the periodic actions (Algorithm 29) the network stays in a valid state in every round afterwards.

3.2 Phase 1: Connect all Heaps by s-edges

In this phase we show that starting from a valid state all existing heaps will eventually be connected by s-edges (defined below), so that they will merge afterwards.
First we give following definitions.

Definition 3.5 *We distinguish between two different kinds of edges that can exist at any time in our network, the edges in the set E and the ones in the set E_s. We say that (x, y) is in E, if $y \in x.N$ and (x, y) in E_s if $y \in x.S$, resulting from a scan from y. We will call the latter ones* s-edges *and denote them by $(x, y)_s$.*

Definition 3.6 *In the directed graph 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 3.7 *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.*

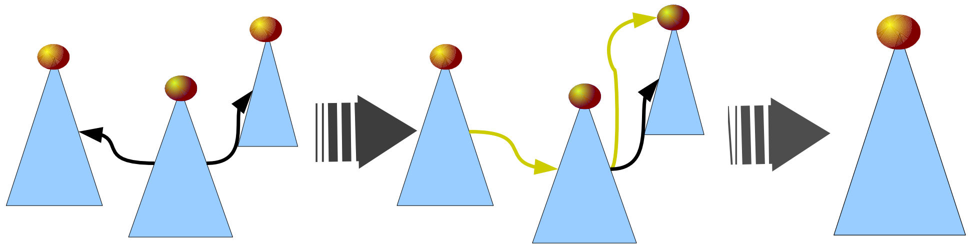


Figure 5.2: The goal of phase 1 of the proof is to show that all heaps are connected through s-edges after $\mathcal{O}(n)$ rounds (middle part of the picture, the light-colored edges depict the s-edges). In phase 2 we show that one heap is been formed after another $\mathcal{O}(n)$ rounds (right part of the picture). From that point on it is not hard to show that a sorted list will be formed as a sub-graph of the network graph and after $\mathcal{O}(n)$ rounds the clique will also be formed (phase 3).

Definition 3.8 *We say that a subset of s-edges $E'_s \subseteq E_s$ is a* s-connectivity set *at round t if all heaps in the graph are s-connected to each other through edges in E'_s at round t.*

In the first phase we will show that after $\mathcal{O}(n)$ rounds all heaps have been connected by s-edges. Let E^0 be the set of edges $(u, v) \in E$ at time $t = 0$. We then show that all these edges are scanned in $\mathcal{O}(n)$ rounds, giving us the connections via s-edges.

Theorem 3.9 *After $\mathcal{O}(n)$ rounds the heaps H_i and H_j connected by $(u, v) \in E^0$ have either merged or been connected by s-edges .*

To prove the theorem we firstly show some basic lemmas needed in the analysis.

Lemma 3.10 *Let $u_1, \cdots u_{|H|}$ be the elements in a heap H in descending order. Then it takes at most i rounds till H is linearized w.r.t u_i.*

Proof. We prove the lemma by induction on the number of rounds i. Note that all nodes are connected by the $x.p$ links only to nodes with larger ids.

Induction base ($i = 0$): The head of the heap is the node with the maximal id therefore trivially, $\forall v \in H - \{h\} : (v.p).id > v.id$ and $\forall v \in H - \{h\}$ with $v.id \geq h.id : (v.p).s = v$.

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

Lemma 3.11 *Once one head learns about the existence of another head, two heaps are merged.*

Proof. Let h_i be the head of heap H_i. Also, let h_j be the head of heap H_j scanning h_i. There can be two cases.

- $h_i < h_j$: In this case, h_i will no longer be a head once h_j scans it and sends it own id.
- $h_i > h_j$: In this case, h_j will no longer be a head and will send an *pred-request* to h_i.

□

In case of a merging of two heaps H_i, H_j, the time it takes until the new heap H is linearized w.r.t. a node u can increase with respect to the linearization time of u in the heap before the merging.

Lemma 3.12 *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 3.10 we know that the linearization time depends on the number of nodes with a larger id in the heap. The number of nodes with a larger id can increase by at most the size of the other heap H_j. Thus, also the linearization time can only increase by at most $|H_j|$. □

From Lemma 3.10 and Lemma 3.12 we immediately get via an inductive argument:

Corollary 3.13 *For any heap H of size $|H|$ in round t it takes at most $|H| - t$ rounds until it forms a sorted list.*

Lemma 3.14 *If a node sends an id with a* forward-from-successor *message, the id will not be delayed by other* forward-from-successor *messages on its way to the head.*

Proof. Once a node sends a message to its predecessor through a *forward-from-successor* message, the number of rounds it takes to reach the head of its heap depends only on the path to the head and the linearization steps. When a node x receives a *forward-from-successor* message, it stores the received id at the head of its neighborhood list $x.N$. So this id will be forwarded immediately, if x is active. If it cannot be forwarded because x is inactive, the node will inform its successor about its inactive state and as a consequence no more *forward-from-successor* messages will be sent to x. That means that no other id can take the place of the one present at the head of $x.N$. So, once x is active again, the id will be sent immediately. □

As a consequence of the observation of Lemma 3.11 we introduce some additional notation to estimate the time it takes until any id is scanned by a head of a heap.

For any edge $(u, v) \in E^0$ 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 in a round t: 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. Corollary 3.13 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* of an id v because if $\phi^t(u, v) = 0$, the id is scanned in round t or has already been scanned by h_i. We then denote by $\Phi^t(u, v) = \min\{\phi^t(w, v) : \mathit{heap}(u) = \mathit{heap}(w)\}$ the minimal delivery time of v for any node in the same heap as u.

For any edge $(u, v) \in E^0$, 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 $(v, h_i)_s$.The following holds:

Lemma 3.15 *If $(u, v) \in E^0$ is an edge between two heaps H_i and H_j, then $\Phi^t(u, v) \leq \max\{2|H_i| + n - t\ , 0\} \leq \max\{3n - t, 0\}$ for all rounds t.*

Proof. We will show the lemma 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($t = 0$):

For any edge $(u, v) \in E^0$ between H_i and H_j let $x \in H_i$ be the node such that $\Phi^0(u, v) = \phi^0(x, v)$. Then $P^0(x) \leq H_i$ as the path length is limited by the number of nodes in the heap, $ID^0(x, v) \leq n$ as not more than n ids are in the system, and following from Lemma 3.13, $LT(x) \leq |H_i|$. Then $\Phi^0(u, v) \leq \phi^0(x, v) \leq 2|H_i| + n \leq 3n$.

Induction step($t \to t'$): For any edge $(u, v) \in E^0$ between H_i and H_j let $x \in H_i$ be the node such that $\Phi^t(u, v) = \phi^t(x, v)$.

Then in round t the following actions can be executed.

- x is inactive and can not forward an id. Then the heap is not linearized w.r.t. x, which implies that the linearization time decreases by one, i.e. $LT^{t'}(x) = LT^t(x) - 1$ and $\phi^{t'}(x, v) = \phi^t(x, v) - 1 \leq 2|H_i| + n - t - 1$ 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 according to Lemma 3.14, x hasn't sent a *forward-from-successor* message with v in a round before, as then there would be another node $y \in H_i$ with $\phi^t(y, v) < \phi^t(x, v)$. Then $ID^{t'}(x, v) \leq ID^t(x, v) - 1$ and $\phi^{t'}(x, v)) = \phi^t(x, v) - 1 \leq 2|H_i| + n - t - 1$.
- 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 $\phi^{t+1}(x.p, v) \leq P^t(x) - 1 + ID^t(x, v) + LT^t(x) = \phi^t(x, v) - 1 \leq 2|H_i| + n - t - 1$

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

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 $x \in H_i$ be the node such that $\Phi^t(u, v) = \phi^t(x, v)$ for an edge $(u, v) \in E^0$ 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, $\Phi^{t+1}(u,v) = \Phi^{t'}(u,v) \leq 2|H_i| + n - t - 1 \leq 3n - (t+1)$.
- Heaps H_i and H_k merge to H_i'. Obviously the length of the path of x can increase and $P^{t+1}(x) \leq P^{t'}(x) + |H_k|$. According to Lemma 3.12 also the linearization time of x can increase and $LT^{t+1}(x) \leq LT^{t'}(x) + |H_k|$. In total $\Phi^{t+1}(u,v) \leq \Phi^{t'}(u,v) + 2|H_k| \leq 2|H_i'| + n - t - 1 \leq 3n - (t+1)$.

Thus, in round $t+1$, $\Phi^{t+1}(u,v) \leq 2|H_i| + n - t - 1 \leq 3n - (t+1)$. □

Hence for every edge $(u,v) \in E^0$ with $u \in H_i$ and $v \in H_j$, $\Phi^t(u,v) = 0$ after $3n$ rounds, which means that the head of H_i scans or has scanned $v \in H_j$ resulting in the s-edge (v, h_i). Thus, we immediately get Theorem 3.9.

3.3 Phase 2: Towards one Heap

Based on the results of Phase 1, we will prove that after $\mathcal{O}(n)$ further rounds a clique is formed. For the purpose of the analysis below, we use the following definitions:

Definition 3.16 *Let $ord(x)$ be the* order *of a node x, i.e. the ranking of the node if we sort all n nodes in the network according to their id (i.e. the node with the largest id m has $ord(m) = 0$, the second largest has order 1, and so on).*

Definition 3.17 *We define the potential $\lambda(x,y)$ of a pair of nodes x and y to be the positive integer equal to $\omega(x,y) = 2 \cdot ord(x) + 2 \cdot ord(y) + K(x,y)$, where $K(x,y) = 1$ if $x.id > y.id$ and 0 otherwise. Also, let for a set of edges $E' \subseteq E$, $\Lambda(E') = \max_{(u,v)\in E'}\{\omega(u,v)\}$, if $E' \neq \emptyset$ and 0 otherwise.*

We proceed by showing the following lemma.

Lemma 3.18 *Two heaps H_i, H_j that are connected by an s-edge $(x,y)_s$ at time t will either stay connected via s-edges $(x_i,y_i)_s$ at time $t+1$ with the property that, $\forall(x_i,y_i)$, the potential $\omega(x_i,y_i)$ of the edges we consider at time $t+1$ is smaller that the potential $\omega(x,y)$ of the edge $(x,y)_s$ we considered at time t, or x and y will be in the same heap.*

Proof. Let $(x,y)_s$ be a s-edge connecting H_i and H_j, i.e. $x \in H_i$, $y \in H_j$. Then according to our algorithm the following actions might be executed.

- x is the head of H_i and $y.id > x.i$ then $y = x.p$ and x sends a *pred-request* message to y, resulting in a merge of H_i and H_j.
- x is the head of H_i and $x.id > y.id$ and y is a new id, then x sends a *scan-ack* to y with its own id and the edge $(y,x)_s$ is created connecting H_i and H_j. Then $\omega(y,x) = 2ord(x) + 2ord(y) + 0 < 2ord(x) + 2ord(y) + 1 = \omega(x,y)$.
- x forwards y to $x.p$ by a *forward-head* message, such that $y \in (x.p).S$ and H_i and H_j are connected by $(x.p,y)_s$. Then $\omega(x.p,y) = 2ord(x.p) + 2ord(y) + K(x.p,y) < 2ord(x) + 2ord(y) + K(x,y) = \omega(x,y)$.

- x receives a new id $z \in x.S$ with $z = \max\{v \in x.N\}$, such that $z.id > y.id$ and $z.id > x.id$. Then x sends a *scan-ack* containing z to y and the s-edge $(x, y)_s$ is substituted by s-edges $(x, z)_s$ and $(y, z)_s$. And H_i and H_j are connected via s-edges. Note that since $(x.p).id > x.id$ and $z.id > x.id, y.id$, $ord(x.p) < ord(x), ord(z) < ord(x)$ and $ord(z) < ord(y)$. The potential of the new edges is: $\omega(x.p, z) = 2ord(x.p) + 2ord(z) + K(x.p, z) < 2ord(x) + 2ord(y) + K(x, y) = \omega_t(x, y)$. $\omega(y, z) = 2ord(y) + 2ord(z) + 0 < 2ord(x) + 2ord(y) + K(x, y) = \omega_t(x, y)$.

- x knows an id $z \in H_k$ with $z = \max\{v \in x.N\}$, $z.id > y.id$ and $z \notin x.S$. Then one of the following cases hold:

  1. $(x, z) \in E^0$, then according to Lemma 3.15 a node u with $u.id > x.id$ and $u \in H_i$ has scanned z resulting in the s-edge $(z, u)_s$ s-connecting H_i and H_k.
  2. x has received z by a *forward-from-predecessor* message. Then a node u with $u.id > x.id$ with $u \in H_i$ has scanned z resulting in the s-edge $(z, u)_s$ s-connecting H_i and H_k.
  3. z was in $x.S$ in a previous round, then the edge $(x, z)_s$ existed s-connecting H_i and H_k.
  4. x has received z by a *forward-from-successor* message. Then there is a node v with $v.id \leq x.id$ in the sub heap rooted at x such that $(v, z) \in E^0$. Then according to Lemma 3.15 a node $w \in H_i$ with $w.id > v.id$ has scanned z and the s-edge $(z, w)_s$ existed s-connecting H_i and H_k. If $w.id > x.id$, H_i and H_k are s-connected by s-edges $(x_i, y_i)_s$ with $\forall(x_i, y_i) : (x.id < w.id < x_i.id \wedge x.id < w.id < y_i.id \wedge z.id \leq x_i.id \wedge z.id \leq y_i.id) \vee (x.id < w.id \leq x_i.id \wedge x.id < w.id \leq y_i.id \wedge z.id < x_i.id \wedge z.id < y_i.id)$. If $w.id < x.id$ then at least as many rounds have passed since w has scanned z as there are nodes on the path from w to x, because z has to be forwarded as many times. Then the edge $(z, w)_s$ has been forwarded or substituted t times or H_i and H_k have merged. Then H_i and H_k are s-connected by s-edges $(x_i, y_i)_s$ with $\forall(x_i, y_i) : (x.id < w.id < x_i.id \wedge x.id < w.id < y_i.id \wedge z.id \leq x_i.id \wedge z.id \leq y_i.id) \vee (x.id < w.id \leq x_i.id \wedge x.id < w.id \leq y_i.id \wedge z.id < x_i.id \wedge z.id < y_i.id)$.

  In each case x sends a *scan-ack* containing z to y and the s-edge $(y, z)_s$ is created. And H_i and H_j are s-connected over s-edges and in all cases the potential shrinks, since for each new s-edge it holds that at least one node is greater and the other node not smaller than the nodes in the edge they replace.

- x is the head of H_i and $x.id < y.id$, then H_i and H_j merge to one heap.

- x is the head of H_i and $x.id > y.id$ and y was in $x.N$ in a previous round, then H_i and H_j are already s-connected by s-edges $(x_i, y_i)_s$ with greater ids by the same arguments as in the case before. Since the ids are greater, the potential shrinks also here.

□

Lemma 3.19 *If E_t is an s-connectivity set at round t, there exists an s-connectivity set E_{t+1} at round $t + 1$ such that $\Lambda(E_{t+1}) < \Lambda(E_t)$.*

Proof. Let E_t be an s-connectivity set a round t. We replace every edge $(x, y)_s \in E_t$ with the edges $(x_i, y_i)_s$ as described in the lemma above. For every pair of heaps that were s-connected at t

through an edge in E_t, there exists a set of s-edges of smaller potential that s-connects the two heaps at $t+1$. We include these edges in E_{t+1}. But at round t all pairs of heaps are s-connected through E_t, which means that at round $t+1$ all pairs of heaps are also s-connected through E_{t+1}. So, E_{t+1} is an s-connectivity set at round $t+1$. Also since all the edges in E_{t+1} have less potential as the ones the replaced in E_t, $\Lambda(E_{t+1}) < \Lambda(E_t)$. □

Theorem 3.20 *After at most 4n+1 rounds, all heaps have been merged into one.*

Proof. From Theorem 3.9 we know that all heaps are s-connected after $\mathcal{O}(n)$ rounds. So after $\mathcal{O}(n)$ rounds there exists the first s-connectivity set, E_0, with $\Lambda(E_0) = \max_{(u,v)\in E_0}\{\omega(u,v)\} = \max_{(u,v)\in E_0}\{2ord(u) + 2ord(v) + K(u,v)\} \leq 2n + 2n + 1 = 4n + 1$. Since for each round t and an s-connectivity set E_t, an s-connectivity set E_{t+1} for round $t+1$ can be found, such that $\Lambda(E_{t+1}) < \Lambda(E_t)$, (i.e. the potential of the s-connectivity set shrinks by every round) after at most $4n+1$ rounds (after the existence of E_s) there exists an s-connectivity set E_∞, such that $\Lambda(E_\infty) = 0$. This means that E_∞ is the empty set. Since E_∞ is an empty s-connectivity set connecting all the heaps of the graph, we know that the graph has only one heap. □

3.4 Phase 3: Sorted List and Clique

Theorem 3.21 *If all nodes form one heap, it takes $\mathcal{O}(n)$ time until the network reaches a legal state.*

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 3.13. Once the heap is linearized and forms a sorted list, each node's id will be sent to the root, the remaining head, after at most n rounds. So the root will be aware of every id. The root, as it sends according to the round-robin process all its information to its successor, will send after n rounds all the ids to it, and the successor will do the same. As a consequence, all nodes will receive all ids at $\mathcal{O}(n)$ rounds. Adding all this together, after $\mathcal{O}(n)$ all nodes will know each other and a clique will be constructed. □

Combining Theorem 3.4, Theorem 3.9, Theorem 3.20 and Theorem 3.21 our main theorem Theorem 3.2 holds.

4 Message Complexity

In this section we give an upper bound for the work spent by each node. We already mentioned that we will distinguish two types of work. The stabilization work, that is spent until a clique is formed, and the maintenance work, that is spent in each round in a legal state. We count the work of a node in the number of messages sent and received.

4.1 Stabilization Work

According to Theorem 3.2 it takes $\mathcal{O}(n)$ rounds to reach a legal state. In each round 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 $\mathcal{O}(n)$ for each node or $\mathcal{O}(n^2)$ in total. By the following lemmas we show that the additional messages sent and received during the linearization are at most $\mathcal{O}(n)$ for each node.

Lemma 4.1 *Each node sends and receives at most $\mathcal{O}(n)$* pred-request, pred-accept *and* new-predecessor *messages during the linearization phase.*

Proof. In each round each node sends at most one *pred-request* and one *pred-accept* message and receives at most one *pred-accept* or *new-predecessor* message. It remains to show that each node receives at most $\mathcal{O}(n)$ *pred-request* and sends at most $\mathcal{O}(n)$ *new-predecessor* messages. Note that it suffices to show that each node receives at most $\mathcal{O}(n)$ *pred-request*, as the number of *new-predecessor* messages directly depends on the number of received *pred-request* messages, to each node, that sends a *pred-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 an id greater than its own id, 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 id of the new predecessor is strictly decreasing. 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 $\mathcal{O}(n)$ *pred-request* and new predecessor messages. □

Lemma 4.2 *Each node sends and receives at most $\mathcal{O}(n)$* scan *and* scan-ack *messages during the linearization phase.*

Proof. Only heads of heaps send scan messages. In each round each head sends exactly one *scan* message. Each scanned node sends a *scanack* message back or stores the id of the head in $x.S$. 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 maximal id in its neighborhood with a *scanack* message, it is scanned at most once by heads with an id smaller then max. By receiving this id the scanning node recognizes, if it is still a head, that it is not the largest id and 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 before the heads stop scanning, because they received a *scanack*. A head that is not the maximal head, that scanned the node so far, will only scan the node one more time and then stop scanning. So a node receives at most $\mathcal{O}(n)$ scan messages from a new maximal head, $\mathcal{O}(n)$ messages from the current maximal head, as each head only sends one scan message per round, and all other scans increase the number of inactive heads, which is limited by $\mathcal{O}(n)$. Regarding the *scanack* messages, since each head scans only once in each round, it receives also at most one *scanack* (that result from sent *scan* messages) message in each round. A node x can also receive a *scanack* message when sending a *scanack* message, but this happen only the if the node to which the *scanack* was sent does not know x, so all in all at most n times. So, all in all, a node receives $\mathcal{O}(n)$ at the whole linearization phase. □

Lemma 4.3 *Each node sends and receives at most $\mathcal{O}(n)$* forward-head *messages through the linearization phase.*

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

4.2 Maintenance Work and Closure

Lemma 4.4 *As soon as the network forms a stable clique with a stable list as a spanning tree, i.e. the network is in a legal state, each node sends and receives at most $\mathcal{O}(1)$ messages in each round, and it stays at a legal state at any round in the future .*

Proof. In a legal state 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 *pred-request* and one *pred-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 *scanack*, sent by the scanned node. Thus, each node sends and receives $\mathcal{O}(1)$ messages in a stable state. Since at no place at the pseudocode any edge is deleted, each node at any future round still maintains at its neighborhood all the nodes in the network and the legal state is maintained. □

5 Single Join and Leave Event

The case of arbitrary churn is hard to analyze formally. Thus, we will show that the clique can efficiently recover considering a single join or leave event in a legal state.

Theorem 5.1 *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 $v.id > u.id$, u sends a *pred-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's id to its predecessor, and after $\mathcal{O}(n)$ rounds the head inserts u to its neighborhood. If $v.id < u.id$ v sends u's id to its predecessor, because it is a new id. Then it takes at most $\mathcal{O}(n)$ rounds until the head receives u's id and scans u, then u assumes the head to be its predecessor and case 1 holds. After $\mathcal{O}(n)$ further rounds each nodes receives u's id and u receives the id of all other nodes in the network. Thus, after $\mathcal{O}(n)$ rounds after a join the nodes form a clique and the sorted list is linearized.

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 *pred-request*, which will be accepted as the node has no other successor. Note that if u is the head of the list, u's successor will recognize that there is no node with a larger id in its neighborhood and will correctly assume to be a head of a list and proceed the scanning. □

6 Outlook

In this chapter we gave a local self-stabilizing time- and work-efficient algorithm that forms a clique out of any weakly connected graph. By forming a clique our algorithm also solves the resource discovery problem, as each node is aware of any other node in the network. Our algorithm is the first algorithm that solves resource discovery in optimal message complexity. Furthermore, our algorithm is self-stabilizing and thus can handle deletions of edges and joining or leaving nodes.

However, our algorithm does not fulfill the requirements in order to incorporate the general framework presented in part 1, which handles node departures. It would be possible to alter the algorithm in order to achieve that, but can probably not be done without affecting the self-stabilization time. However, this assumption has not been proven formally and would be interesting to investigate in the future.

Bibliography

[1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications, ACM*, 17:643-644, 1974.

[2] I Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions of Networking*, 11(1):17–32, 2003.

[3] A. Berns, S. Ghosh, and S.V. Pemmaraju. Brief announcement: a framework for building self-stabilizing overlay networks. In *PODC'10*, pages 398–399, 2010.

[4] L. Blin, S. Dolev, M. Gradinariu Potop-Butucaru, and S. Rovedakis. Fast self-stabilizing minimum spanning tree construction - using compact nearest common ancestor labeling scheme. In *DISC'10*, pages 480–494, 2010.

[5] L. Blin, S. Dolev, M. Gradinariu Potop-Butucaru, and S. Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *Journal of Parallel Distributed Computing*, 71(3):438–449, 2011.

[6] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *FOCS*, pages 383–395, 1985.

[7] T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list. In *SSS '08*, pages 124-140, 2008.

[8] T. Hérault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A model for large scale self-stabilization. In *IPDPS*, pages 1–10, 2007.

[9] A.G. Myasnikov, V. Shpilrain, and A. Ushakov. *Group-based cryptography*. Advanced courses in mathematics, 2008.

[10] M.C. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[11] H.V. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *OPODIS*, pages 88–102, 2005.

[12] C. Scheideler and S. Schmid. A distributed and oblivious heap. In *ICALP (2)*, pages 571–582, 2009.

[13] B. Haeupler, G. Pandurangan, D. Peleg, R. Rajaraman, Z. Sun. Discovery through Gossip. In *SPAA*, 2011.

[14] M. Nesterenko, S. Tixeuil. Discovering Network Topology in the Presence of Byzantine Faults. In *IEEE Transactions on Parallel and Distributed Systems*, pp. 1777-1789, 2009.

[15] H. Abu-amara,B. A. Coan,S. Dolev,A. Kanevsky,J. L. Welch K. Self-stabilizing topology maintenance protocols for high-speed networks. In *IEEE/ACM Transactions on Networking* Volume 4 Issue 6, Pages 902 - 912 , 1996.

[16] M. Harchol-Balter, T. Leighton, D. Lewin. Resource discovery in distributed networks. In PODC '99, pages 229-237, 1999

[17] S. Kutten, D. Peleg, U. Vishkin. Deterministic resource discovery in distributed networks. In *SPAA '01*, pages 77-83, 2001.

[18] K. M. Konwar, D. Kowalski, A. A. Shvartsman. Node discovery in networks In *Journal of Parallel and Distributed Computing 69, 4 (April 2009)*, pages 337-348, 2009.

[19] C. Schindelhauer, G. Schomaker. Weighted distributed hash tables. *SPAA '05*, pages 218 - 227, 2005.

[20] G. Giakkoupis, V. Hadzilacos. A Scheme for Load Balancing in Heterogenous Distributed Hash Tables In *PODC '05*, 2005

[21] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform distribution requirements. In *SPAA '02*, pages 53-62, 2002.

[22] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *SPAA '00*, pages 119-128, 2000.

[23] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC '97*, pages 654-663 , 1997.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[25] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001

[26] C. Cramer and T. Fuhrmann. Self-stabilizing ring networks on connected graphs. In *Technical report*, University of Karlsruhe (TH), Fakultaet fuer Informatik, 2005.

[27] S. Dolev and R. I. Kat. HyperTree for self-stabilizing peer-to-peer systems. In *Distributed Computing*, 20(5), pages 375–388, 2008.

[28] S. Dolev and N. Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theoretical Computer Science*, 410(6-7):514–532, 2009.

[29] S. Dolev and N. Tzachar. Spanders: distributed spanning expanders. In *SAC '10*, pages 1309–1314, 2010.

[30] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC '09*, pages 131–140, 2009.

[31] Matthias Feldotto, Christian Scheideler, Kalman Graffi HSkip+: A self-stabilizing overlay network for nodes with heterogeneous bandwidths In *P2P 2014*,p. 1-10, 2014.

[32] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid. A self-stabilizing and local delaunay graph construction. In *ISAAC '09*, pages 771–780, 2009.

[33] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. Re-chord: a self-stabilizing chord overlay network. In *SPAA '11*, pages 235–244, 2011.

[34] R. Nor, M. Nesterenko, and C. Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *SSS '11*, pages 356–370, 2011.

[35] M. Onus, A. W. Richa, and C. Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *ALENEX '07* pages 99–108, 2007.

[36] D. Gall, R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *LATIN*, pages 294–305, 2010.

[37] A. Shaker and D. S. Reeves. Self-stabilizing structured ring topology p2p systems. In *P2P '05*, pages 39–46, 2005.

[38] N. Harvey. CPSC 536N: Randomized Algorithms, Lecture 3. In *University of British Columbia*, Pages 5, 2011-12.

[39] T. Ernvall, S. El Rouayheb, C. Hollanti, H. Vincent Poor Capacity and Security of Heterogeneous Distributed Storage Systems In *CoRR*,2012

[40] H. Shena, C.-Z. Xub. Hash-based proximity clustering for efficient load balancing in heterogeneous DHT networks. In *J. Parallel Distributed Computing 68*, 686-702, 2008.

[41] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp and I. Stoica. Load balancing in structured P2P systems. In *IPTPS 03*, 2003.

[42] Q. Yu, C. Wan Sung, T. H. Chan Repair topology design for distributed storage systems In *ICC 12*, Pages: 7009 - 7013, 2012

[43] P. B. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *IEEE INFOCOM*, 2005.

[44] M. Bienkowski, A. Brinkmann, M. Klonowski and M. Korzeniowski, Miroslaw. SkewCCC+: a heterogeneous distributed hash table. In *OPODIS'10*, pages 219-234, 2010.

[45] J. R. Santos and R. Muntz. Performance Analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In *ACM Multimedia Conference '98*, p. 303-308, 1998.

[46] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann and T. Cortes. Reliable and randomized data distribution strategies for large scale storage systems. In *HiPC '11*, p. 1-10, 2011.

[47] S.-Y. Didi Yao, C. Shahabi, and R. Zimmermann. BroadScale: Efficient scaling of heterogeneous storage systems. In *Int. J. on Digital Libraries, vol. 6*, pages 98-111, 2006.

[48] A. Brinkmann, S. Effert, F. Meyer auf der Heide and C. Scheideler. Dynamic and Redundant Data Placement. In *ICDCS '07*, pp.29, 2007.

[49] T. Cortes and J. Labarta. Taking advantage of heterogeneity in disk arrays. In *J. Parallel Distributed Computing 63*, pages 448-464, 2003.

[50] M. Mense and C. Scheideler. SPREAD: An adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems In *SODA '08*, 2008.

[51] J. Aspnes and G. Shah. Skip graphs. In *SODA '03*, pages 384–393, 2003.

[52] B. Awerbuch and C. Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *SODA '04*, pages 318–327, 2004.

[53] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting. In *SPAA '04*, p. 170-179.

[54] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: a scalable overlay network with practical locality properties. In *USITS'03*, pages 9-9, 2003.

[55] F. Kuhn, S. Schmid, and R. Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *IPTPS '05*, pages 13–23, 2005.

[56] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC '02*, pages 183-192, 2002.

[57] M. Naor and U. Wieder. Novel architectures for p2p applications: The continuous-discrete approach. *ACM Transactions on Algorithms*, 3(3), 2007.

[58] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97*, pages 311-320, 1997.

[59] K. Albrecht, F. Kuhn, and R. Wattenhofer. Dependable peer-to-peer systems withstanding dynamic adversarial churn. In *Dependable Systems*, pages 1–8, 2006.

[60] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, pages 131–145, New York, NY, USA, 2001. ACM.

[61] D. Angluin, M. J. Fischer, and H. Jiang. Stabilizing consensus in mobile networks. In *DCOSS*, pages 37–50, 2006.

[62] B. Awerbuch and C. Scheideler. Towards scalable and robust overlay networks. In *IPTPS*, 2007.

[63] B. Awerbuch and C. Scheideler. Towards a scalable and robust dht. *Theory of Computing Systems*, 45(2):234–260, 2009.

[64] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *IPDPS 09*, pages 1–8. IEEE, 2009.

[65] S. Bianchi, A. Datta, P. Felber, and M. Gradinariu. Stabilizing peer-to-peer spatial filters. In *ICDCS*, page 27, 2007.

[66] E. Caron, Fr. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.

[67] T. Deepak Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[68] B. Doerr, L. A. Goldberg, L. Minder, T. Sauerwald, and C. Scheideler. Stabilizing consensus with the power of two choices. In *SPAA*, pages 149–158, 2011.

[69] D. Dolev, E. Hoch, and R. van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *PODC*, volume 4878, 2007.

[70] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[71] D. Foreback, A. Koutsopoulos, M. Nesterenko, C. Scheideler, and T. Strothmann. On stabilizing departures in overlay networks. In *SSS 2014*, pages 48–62, 2014.

[72] T. P. Hayes, J. Saia, and A. Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, 25(4):261–278, 2012.

[73] T. Hérault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. Brief announcement: Self-stabilizing spanning tree algorithm for large scale systems. In *SSS*, pages 574–575, 2006.

[74] B. Javadi, D. Kondo, J.-M. Vincent, and D. P. Anderson. Mining for statistical models of availability in large-scale distributed systems: An empirical study of seti@home. In *MASCOTS 2009*, pages 1–10, 2009.

[75] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. A self-stabilization process for small-world networks. In *IPDPS 2012*, pages 1261–1271, 2012.

[76] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. CONE-DHT: A distributed self-stabilizing algorithm for a heterogeneous storage system. *DISC 13*, abs/1307.6747, 2013.

[77] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. A deterministic worst-case message complexity optimal solution for resource discovery. *Theoretical Computer Science*, 584:67–79, 2015.

[78] A. Koutsopoulos, C. Scheideler, and T. Strothmann. Towards a universal approach for the finite departure problem in overlay networks. In *SPAA 15*, pages 77–79, 2015.

[79] F. Kuhn, S. Schmid, and R. Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Distributed Computing*, 22(4):249–267, 2010.

[80] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Analysis of the evolution of peer-to-peer systems. In Aleta Ricciardi, editor, *PODC 02*, pages 233–242, 2002.

[81] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel Distributed Computing*, 62(5):766–791, 2002.

[82] R. Mohd Nor, M. Nesterenko, and S. Tixeuil. Linearizing peer-to-peer systems with oracles. Technical Report TR-KSU-CS-2012-02, Dept. of Computer Science, Kent State University, 2012.

[83] J. Saia and A. Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *IPDPS*, pages 1–12, 2008.

[84] C. Scheideler. How to spread adversarial nodes?: rotate! In *STOC*, pages 704–713, 2005.

[85] wikipedia. Apache hadoop.

[86] wikipedia. Bittorrent.

[87] P. Zave. Using lightweight modeling to understand chord. *Computer Communication Review*, 42(2):49–57, 2012.

[88] B. Leong, B. Liskov, E. D. Demaine, *EpiChord: Parallelizing the Chord lookup algorithm with reactive routing state management* ICON 04, Pages 1243-1259, 2004

[89] V. A. Mesaros, B. Carton, P. Van Roy, P. Sainte Barbe, *S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord*, PDPTA 02, pages 23-26, 2002

[90] J. Jiang, R. Pan, C. Liang, W. Wang, BiChord: An Improved Approach for Lookup Routing in Chord ADBIS, 2005, pages 338-348,

[91] J. Wang, Z. Yu. *A new variation of Chord with novel improvement on lookup locality*, (GCA 06), Pages 18-24, 2006.

[92] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on Demand (P2P ’05). 2005