Re-Chord: A Self-stabilizing Chord Overlay Network*

Sebastian Kniesburges, Andreas Koutsopoulos, Christian Scheideler
University of Paderborn
Paderborn, Germany
seppel@upb.de, koutsopo@mail.upb.de, scheideler@mail.upb.de

ABSTRACT

The Chord peer-to-peer system is considered, together with CAN, Tapestry and Pastry, as one of the pioneering works on peer-to-peer distributed hash tables (DHT) that inspired a large volume of papers and projects on DHTs as well as peer-to-peer systems in general. Chord, in particular, has been studied thoroughly, and many variants of Chord have been presented that optimize various criteria. Also, several implementations of Chord are available on various platforms. Though Chord is known to be very efficient and scalable and it can handle churn quite well, no protocol is known yet that guarantees that Chord is self-stabilizing, i.e., the Chord network can be recovered from any initial state in which the network is still weakly connected. This is not too surprising since it is known that in the Chord network it is not locally checkable whether its current topology matches the correct topology. We present a slight extension of the Chord network, called Re-Chord (reactive Chord), that turns out to be locally checkable, and we present a self-stabilizing distributed protocol for it that can recover the Re-Chord network from any initial state, in which the n peers are weakly connected, in $\mathcal{O}(n \log n)$ communication rounds. We also show that our protocol allows a new peer to join or an old peer to leave an already stable Re-Chord network so that within $\mathcal{O}((\log n)^2)$ communication rounds the Re-Chord network is stable again.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms, network problems*; E.1 [**Data**]: Data Structures—*Distributed Data Structures*; C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*Distributed networks*

General Terms

Algorithms, Theory, Reliability

Keywords

Chord, peer-to-peer networks, self-stabilizing protocols

1. INTRODUCTION

Peer-to-peer systems have received a lot of attention in the past years as they have many interesting applications including social networks, file sharing, streaming, instant messaging or VoIP. In research, the pioneering and most influential systems are usually considered to be Chord [28], CAN [24], Pastry [25] and Tapestry [30]. The networks of these systems have in common that they have a low diameter and degree while being quite robust to churn. However, no self-stabilizing protocol is known for any of these four, i.e., no distributed protocol is known for these that can recover the desired topology from any weakly connected state. Self-stabilization is important as unusually high churn, network partitions or adversarial behavior may push these networks into a state from which they cannot recover using the known protocols. In this paper we present Re-Chord, a self-stabilizing variant of the Chord network [28]. We will show that efficient self-stabilization is possible for Re-Chord while maintaining the advantages of the Chord network.

1.1 The Chord network and its variants

The Chord system was introduced in an influential paper by Stoica, Morris, Karger, Kaashoek and Balakrishnan [28]. Chord is basically a combination of a hypercubic network with an indexing method called consistent hashing [16]. 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 $h : U \rightarrow [0, 1)$ (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 $1 \leq i \leq m$, so that $h(v) + 1/2^m (\text{mod}\, 1) \leq h(succe-ssor(v)) \leq h(v) + 1/2^{m-1} (\text{mod}\, 1)$. 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 so 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

[*] Partially supported by DFG grant SCHE 1592/1-1.

SPAA'11, June 4–6, 2011, San Jose, California, USA.

presentation of the Chord network. In [18] a variant called EPI Chord is presented that allows the system to do parallel searches for the best route to the node storing the data for a certain search key. This does not improve the asymptotical worst-case cost of $\mathcal{O}(\log n)$ messages of Chord but it can achieve $\mathcal{O}(1)$ hop lookup performance under lookup intensive workloads due to caching. In [20] 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 [15] and [29], where links to the predecessors are stored instead of only links to the successors of a node. In [29] 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 [21] 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.

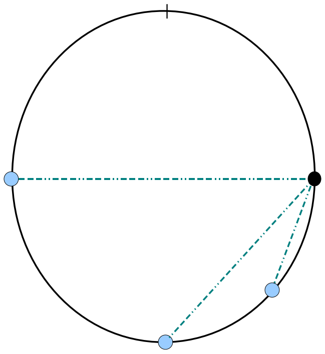

Figure 1: A real node (black) has its virtual nodes (fingers) at distance $1/2^k$ away from itself, at the clockwise direction.

1.2 Other related work

There is a large body of literature on how to maintain peer-to-peer networks efficiently, e.g., [1, 2, 4, 25, 11, 17, 19, 22, 24, 28, 26]. While many results are already known on how to keep an overlay network in a legal state, not much is known about self-stabilizing overlay networks. In the field of self-stabilization, researchers are interested in algorithms that are guaranteed to eventually converge to a desirable system state from any initial configuration. The idea of self-stabilization in distributed computing first appeared in a classical paper by E.W. Dijkstra in 1974 [8] in which he looked at the problem of self-stabilization in a token ring. Since Dijkstra's paper, self-stabilization has been studied in many contexts, including communication protocols, graph theory problems, termination detection, clock synchronization, and fault containment. For a survey see, e.g., [5, 9, 12].

Interestingly, though self-stabilizing distributed computing has received a lot of attention for many years, the problem of designing self-stabilizing networks has attracted much less attention. The universal techniques known for distributed computing in static networks (like logging) are not applicable here as they have not been designed to actively perform local topology changes (network changes are only considered as faults or dynamics not under the control of the algorithm). In order to recover scalable overlays from any initial graph, researchers have started with simple non-scalable line and ring networks. The Iterative Successor Pointer Rewiring Protocol [7] and the Ring Network [27] organize the nodes in a sorted ring. In [23], Onus et al. present a local-control strategy called linearization for converting an arbitrary connected graph into a sorted list. Clouser et al. [6] formulate a variant of the linearization technique for asynchronous systems in order to design a self-stabilizing skip list. Gall et al. [10] discuss models that capture the parallel time complexity of locally self-stabilizing networks that avoids bottlenecks and contention. Jacob et al. [14] generalize insights gained from graph linearization to two dimensions and present a self-stabilizing construction for Delaunay graphs. In another paper, Jacob et al. [13] 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 [3] the authors present a general framework for the self-stabilizing construction of any overlay network. However, the algorithm requires the knowledge of the 2-hop neighborhood for each node and involves the construction of a clique. In that way, failures at the structure of the overlay network can easily be detected and repaired.

1.3 Our contributions

In this paper we present Re-Chord, a self-stabilizing variant of Chord. 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. *Re-Chord stabilizes after $O(n \log n)$ rounds from any weakly connected state w.h.p. The final state of Re-Chord contains Chord as a subgraph, so it can faithfully emulate any applications on top of Chord.*

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

1.4 Organization of the Paper

The paper is organized as follows: In Section 2 a formal presentation of the self-stabilization rules is given. In Section 3 we prove that our rules indeed lead the network into a stable state, and in Section 4 we analyze the steps needed for the network to recover after a peer joins or leaves the network. In Section 5 we present our simulation results and, finally, in Section 6 we derive our conclusions.

2. THE RE-CHORD NETWORK

2.1 Our model

We model the overlay network as a directed graph $G = (V, E)$ where $|V| = n$. Each node is assumed to have a unique identifier, a real number in $[0, 1)$ that is immutable. For simplicity, we assume that time proceeds in synchronous rounds, and all messages generated in round i are delivered simultaneously at the end of round i. So we are using the standard synchronous message-passing model. In each round, each node can only inspect its own state. Beyond that, a node does not know anything, including the current size of the overlay network. Only local topology changes are allowed, i.e., a node may decide to cut a link to a neighbor or ask two of its neighbors to establish a link. The decisions to cut or establish links are controlled through actions (which we will also call rules) that we define more precisely later in this section.

When using the synchronous message-passing model, the global state of the system at the beginning of each round is well-defined. A computation is a sequence of states such that for each state s_i

at the beginning of round i, the next state s_{i+1} is obtained after executing all actions that were fired in round i and receiving all messages that they generated. We call a distributed algorithm *self-stabilizing* if from any initial state in which the overlay network is weakly connected, i.e. it forms a weakly connected directed graph (so that a legal state is still reachable), it eventually reaches a legal state in which no more state changes are taking place in the nodes. In our context, a legal state corresponds to the desired Re-Chord topology.

2.2 State of Re-Chord

In the 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. In the following $u = u_{id}$. For the *self-stabiliza-tion* process every *real* node simulates a number of *virtual nodes*. A simulated virtual node u_i belonging to a real node u has the identifier $u_i = u + \frac{1}{2^i} \mod 1$. We further define $u_0 = u$. The virtual nodes belonging to the same real node are called siblings. Given a node u, we define $m \in \mathbb{N}$ to be the maximal value such that u has no outgoing edge to a real node that is in the interval $\left[u_0, u + \frac{1}{2^m}\right]$. Then u_m is the virtual node with the smallest distance to u. In the stable Re-Chord network each node (virtual or real) has a connection to its closest left (smaller) and closest right (larger) node, as well as to closest left and closest right *real* node among all nodes in the system.

To describe the Re-Chord network and the corresponding self-stabilizing algorithm we need the following notation:

- The graph consists of three different kinds of edges: E_u denotes the set of *unmarked edges*, E_c the set of *connection edges* and E_r the set of *ring edges*. Let $E = E_u \cup E_c \cup E_r$. The graph can be a multi-graph, i.e. an edge (u, v) can be in E more than once due to different markings u, c, r.

- The graph consists of two different kinds of nodes V_r and V_v, where V_r denotes all real nodes and V_v denotes all virtual nodes. Let $V = V_r \cup V_v$.

- Let $[u, v]$ be the interval from u to v that contains all nodes w with identifiers $u < w < v$, for the case $u < v$, and identifiers w for which $w < v$ or $u < w$ for the case $u > v$. E.g. $0, 2 \in [0.8, 0.3]$, but $0.2 \notin [0.3, 0.8]$.

- Let $N_u(u_i) = \{v \in V | (u_i, v) \in E_u\}$ be the unmarked neighborhood of a node u_i (virtual or real). Let $N_r(u_i)$ and $N_c(u_i)$ be the neighborhoods given by the outgoing ring or connection edges of u_i.

- Let $S(u_i) = \{u_0, u_1, \cdots u_m\}$ be the set of siblings of a node u_i.

- Let $N(u_i) = S(u_i) \cup (\bigcup_{0 \leq j \leq m} N_u(u_j))$ be the known neighborhood of node u_i, due to the unmarked edges only.

Note that $V_r \cap N(u_0) \neq \varnothing$ at any point in time since $u_0 \in N(u_0)$. The virtual nodes and edge sets E_u, E_r and E_c are needed for the self-stabilization process and are computed internally by every real node (peer). The final Re-Chord network is built on top of this internal graph. The Re-Chord network is a network on the real nodes. The edges in the Re-Chord network are defined by

$$E_{Re-Chord} = \left\{(u, v) \in V_r^2 : \exists i, (u_i, v) \in E_u \cup E_r\right\}$$

Chord has two kinds of edges, successor-predecessor edges that form the Chord ring, as well as fingers. In the stable state each real node in Re-Chord has an edge to its closest right and closest left real neighbor (which would be the successor and predecessor of that node in Chord), so these edges simulate the successor-predecessor edges. In Chord, each node u has a finger edge, which connects the node with the node being the closest successor of $u + \frac{1}{2^i}(\mod 1)$ (formally described in section 1.1), in a clockwise direction along a $[0, 1)$ circle, for different values of i. Re-Chord achieves the same as each real node u creates a virtual node having value $u + \frac{1}{2^i}(\mod 1)$. Since this particular virtual node will be connected to the real node being the closest successor to $u + \frac{1}{2^i}(\mod 1)$, this leads to the same connection as in the Chord network. Therefore, each edge of Chord is included in Re-Chord, which implies the following fact.

FACT 2.1. *In the stable state, Chord is a subgraph of Re-Chord.*

So, for each connection in Chord there is a virtual node in Re-Chord. Since each node in Re-Chord (virtual or real) has at most 4 outgoing unmarked edges (two to their closest left and right neighbors, as well as two edges to their closest left and right real neighbors) it holds that $|E_u \cup E_r| \leq 4|E_{Chord}|$, where E_{Chord} is the set of edges of Chord. We also use connection edges, which do not participate in the routing, but only serve for the self-stabilization process. As we will see, each virtual node generates $\Theta(\log(n))$ connection edges in expectation, and since the number of nodes in Re-Chord are $O(n \log(n))$ w.h.p., the expected number of connection edges is $O(n \log^2(n))$.

2.3 Self-Stabilization Rules

In the following we will define the distributed algorithm by formulating the rules carried out by every node. For each rule we will give a short informal description. and a formal definition as a set of actions. An action has the form:

$$< name >:< guard >\rightarrow< commands >$$

The $< name >$ is the label of the action, $< guard >$ is a Boolean predicate over variables of the node and the term $< commands >$ is a sequence of commands that may involve any of the variables of the executing node or its neighbors [10]. A $command$ can be a direct assignment. In addition, we introduce the notion $A \leftarrow B$, where A and B are sets and $\leftarrow$ can be interpreted as a "delayed" := (assignment). That means that this assignment will only be executed right before the next round.

Note that these rules are all applied for all combinations of parameters in one round and in the order in which they are presented below, in each node (although a parallel application will not violate the correctness). In addition, if node v inserts an edge (u, w) between its neighbors $u, w \in N(v)$, then u is only aware of that edge in the next round. On the other hand, if a node v deletes an edge (v, w) the edge will not be considered in the rules for v for the rest of the same round. Note also that the rules are based on local knowledge.

Before a node applies the set of rules, it updates its variables by computing a new m, as defined above, and the new neighborhoods.

1. Virtual Nodes: Create all virtual nodes u_i, $i \leq m$ (if not existing). Delete all virtual nodes u_j, $j > m$ (if existing) as they are needless. In case a virtual node u_i is deleted, the virtual node u_m is informed about u_i's neighborhood.

   - $create - virtualnodes(u) : u_i \notin S(u) \wedge i \leq m \rightarrow S(u) := S(u) \cup \{u_i\}$
   - $delete - virtualnodes(u) : u_i \in S(u) \wedge i > m \rightarrow S(u) := S(u)/\{u_i\}, N_u(u_m) := N_u(u_m) \cup N_u(u_i) \cup N_r(u_i) \cup N_c(u_i)$

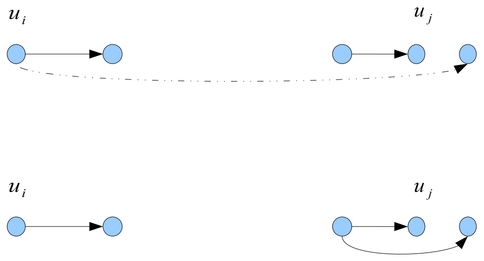

Figure 2: Nodes before and after the application of the overlapping neighborhood rule. The dotted line is an overlapping edge. After the rule the node is reassigned to another neighborhood

2. Overlapping Neighborhood: Let u be a real node. For each u_i check the neighborhood $N_u(u_i)$. If there is a $w \in N_u(u_i)$ and a $u_j \in S(u_i)$ such that $w < u_j < u_i$ or $w > u_j > u_i$, then replace (u_i, w) by (u_j, w). This is done, because u_j is closer to w and u_i is aware of this fact as u_i and u_j belong to the same real node (See Fig 2).

   - $check - all - neighborhoods(u) : u_i \in S(u) \rightarrow check - neighborhood(u_i)$
   - $check - neighborhood(u_i) : w \in N_u(u_i) \wedge u_j \in S(u_i) \wedge (w < u_j < u_i \vee w > u_j > u_i) \rightarrow N_u(u_j) := N_u(u_j) \cup \{w\}, N_u(u_i) := N_u(u_i)/\{w\}$

3. Closest Real Neighbor: For each u_i find the closest left and right real neighbor. Inform all neighbors in the interval between the closest real neighbors about the found closest real neighbors. We also define the closest left and right real nodes of u_i as $r_l(u_i) = \max\{w \in N(u_i) : w \in V_r \wedge w < u_i\}$ and $r_r(u_i) = \min\{w \in N(u_i) : w \in V_r \wedge w > u_i\}$.

   - $all - realneighbors(u) : u_i \in S(u) \rightarrow left - realneighbor(u_i), right - realneighbor(u_i)$
   - $left - realneighbor(u_i)$: $v = \max\{w \in N(u_i) : w \in V_r \wedge w < u_i\}, y \in N_u(u_i), y > u_i \vee v < y < u_i, v > r_l(y) \rightarrow N_u(u_i) := N_u(u_i) \cup \{v\}, N_u(y) \leftarrow N_u(y) \cup \{v\}, r_l(u_i) := v$
   - $right - realneighbor(u_i)$: $v = \min\{w \in N(u_i) : w \in V_r \wedge w > u_i\}, y \in N_u(u_i), y < u_i \vee v > y > u_i, v < r_r(y) \rightarrow N_u(u_i) := N_u(u_i) \cup \{v\}, N_u(y) \leftarrow N_u(y) \cup \{v\}, r_r(u_i) := v$

4. Linearization: For each u_i do: Sort all $w \in N_u(u_i), w < u_i$ in descending order and create edges (w_l, w_{l+1}). Sort all $w \in N_u(u_i), w > u_i$ in ascending order and create edges (w_l, w_{l+1}). We call this forwarding of an edge, because the starting point of an edge is moved to a node closer to its endpoint. Create backward edges from the closest neighbors to u_i. We call this mirroring of an edge. Note: When the mirroring rule is executed, u_i has only its two closest (left and right) neighbors, by rule 3.

   - $linearize - all(u) : u_i \in S(u) \rightarrow lin - left(u_i), lin - right(u_i), mirroring(u_i)$
   - $lin - left(u_i) : w, v \in N_u(u_i) \wedge v, w < u_i \wedge v = \max\{y \in N_u(u_i) : y < w\}) \rightarrow N_u(w) \leftarrow N_u(w) \cup \{v\}, N_u(u_i) := N_u(u_i)/\{v\}$
   - $lin - right(u_i) : w, v \in N_u(u_i) \wedge v, w > u_i \wedge v = \min\{y \in N_u(u_i) : y > w\} \rightarrow N_u(w) \leftarrow N_u(w) \cup \{v\}, N_u(u_i) := N_u(u_i)/\{v\}$
   - $mirroring(u_i) : v \in N(u_i) \rightarrow N_u(v) \leftarrow N_u(v) \cup \{u_i\}, N_u(u_i) := N_u(u_i) \cup \{r_l(u_i)\}, N_u(u_i) := N_u(u_i) \cup \{r_r(u_i)\}$

5. Ring Edge: By the linearization rule only a sorted list can be achieved. We need further rules to close the ring. We establish special marked ring edges E_r to do so. These edges are created if a node misses a right or left neighbor and assumes to be the node of maximal or minimal identifier in $[0, 1)$. The edges are directed to the node missing a neighbor, so are outgoing edges and can be forwarded by the nodes assumed to be the minimal/maximal node. For each u_i do: if the node has no right (resp. left) neighbor create a special ring edge from the smallest (resp. largest) known node $x \in N(u_i)$ to u_i. If u_i has such an outgoing ring edge, say to node w, and $w > u_i$ (resp. $w < u_i$) then create an unmarked edge (x, w) with $x \in N(u_i) \cup N_r(u_i), u_i < w < x$ (resp. $x < w < u_i$). If there is no such x create the ring edge (v, w) to the smallest (resp. largest) known $v \in N(u)$. If a x or v can be found, delete the ring edge (u_i, w).

   - $create - all - ring - edges(u) : u_i \in S(u) \rightarrow create - ring - edge - left(u_i), create - ring - edge - right(u_i)$
   - $create - ring - edge - left(u_i)$: $v = \max\{x \in N(u)\} \wedge \nexists w \in N_u(u_i) : w < u_i \rightarrow N_r(v) \leftarrow \{u_i\} \cup N_r(v)$
   - $create - ring - edge - right(u_i)$: $v = \min\{x \in N(u)\} \wedge \nexists w \in N_u(u_i) : w > u_i \rightarrow N_r(v) \leftarrow \{u_i\} \cup N_r(v)$
   - $forward - all - ring - edges(u) : u_i \in S(u) \rightarrow forward - ring - edge - l1(u_i), forward - ring - edge - l2(u_i), forward - ring - edge - r1(u_i), forward - ring - edge - r2(u_i)$
   - $forward - ring - edge - l1(u_i) : w \in N_r(u_i) \wedge w > u_i \wedge v = \min\{x \in N(u_i)\} \wedge v \neq u_i \wedge \nexists x \in N(u_i) \cup N_r(u_i) : x > w \rightarrow N_r(v) \leftarrow \{w\} \cup N_r(v), N_r(u_i) := N_r(u_i)/\{w\}$
   - $forward - ring - edge - l2(u_i) : w \in N_r(u_i) \wedge w > u_i \wedge \exists x \in N(u_i) \cup N_r(u_i) : x > w \rightarrow N_u(x) \leftarrow \{w\} \cup N_u(x), N_r(u_i) := N_r(u_i)/\{w\}$
   - $forward - ring - edge - r1(u_i) : w \in N_r(u_i) \wedge w < u_i \wedge v = \max\{x \in N(u_i)\} \wedge v \neq u_i \wedge \nexists x \in N(u_i) \cup N_r(u_i) \wedge x < w \rightarrow N_r(v) \leftarrow \{w\} \cup N_r(v), N_r(u_i) := N_r(u_i)/\{w\}$
   - $forward - ring - edge - r2(u_i) : w \in N_r(u_i) \wedge w < u_i \wedge \exists x \in N(u_i) \cup N_r(u_i) \wedge x < w \rightarrow N_u(x) \leftarrow \{w\} \cup N_u(x), N_r(u_i) := N_r(u_i)/\{w\}$

6. Connection Edges: We introduce another set of edges, the connection edges, which are used to ensure that all nodes are in one connected component closing possible gaps between

contiguous virtual siblings. For all neighbored virtual nodes u_i, u_j, i.e. $u_i < u_j = \min\{u_l : u_l > u_i\}$, connection edge between u_i, u_j is created. If a node u_i has an outgoing connection edge (u_i, x) it creates a new connection edge (w, x) with $w = \max\{v \in N_u(u_i) \cup S(u_i)\}$. If such an w does not exist, u_i creates a (unmarked) backward edge (x, u_i).

- $connect{-}virtual{-}nodes(u) : u_i, u_j \in S(u) \wedge u_j = \min\{u_l \in S(u), u_l > u_i\} \rightarrow N_c(u_i) := N_c(u_i) \cup \{u_j\}$
- $forward{-}all{-}cedges(u) : u_i \in S(u) \rightarrow forward{-}cedges - 1(u_i), forward - cedges - 2(u_i)$
- $forward - cedges - 1(u_i) : v \in N_c(u_i) \wedge w = \max\{x \in N_u(u_i) \cup S(u_i) : x < v\} \wedge w \neq u_i$ $\rightarrow N_c(w) \leftarrow N_c(w) \cup \{v\}, N_c(u_i) := N_c(u_i)/\{v\}$
- $forward - cedges - 2(u_i) : v \in N_c(u_i) \wedge u_i = \max\{x \in N_u(u_i) \cup S(u_i) : x < v\} \wedge w = u_i$ $\rightarrow N_u(v) \leftarrow N_u(v) \cup \{u_i\}, N_c(u_i) := N_c(u_i)/\{v\}$

3. ANALYSIS

We will frequently need the following result, which follows from standard techniques.

LEMMA 3.1. *The number of virtual nodes between two real nodes are no more than $c \log n$, where c is a constant, w.h.p.. The total number of nodes in the network is $\Theta(n \log n)$ w.h.p.*

3.1 Correctness

We will show the correctness of the algorithm given by the rules by proving our main theorem. For this we will divide the self-stabilization process into different phases and determine the correctness and running time of each phase. In our proof we will assume that the phases finish one after the other, though this does not restrict the general case, as the resulting properties of this phase hold forever once established.

3.1.1 Phase 1:Connection

First we want to ensure that all virtual and real nodes belong to the same connected component formed by unmarked edges. In the initial state the graph formed by the real nodes is weakly connected , i.e. there is an edge (u, v) in the graph given by the real nodes, if there is an edge $(u_i, v_j) \in E_r \cup E_u \cup E_c$. However the initial graph given by the virtual (including the real) nodes does not have to be weakly connected as there might be nodes u_i, u_j that are not connected. Note that this is the only case that the graph of virtual nodes is not weakly connected. We will show:

LEMMA 3.2. *After $\mathcal{O}(n \log n)$ rounds all nodes are weakly connected by unmarked edges, i.e. there is path of unmarked edges, which can be traversed in both directions, for each pair of nodes connecting them. Two contiguous virtual siblings u_i, u_j are connected by unmarked edges over nodes w with $u_i < w < u_j$.*

We will prove the lemma by proving three claims, that show that the graph becomes weakly connected by connecting all u_i, u_j and if it is weakly connected it will become weakly connected by unmarked edges.

CLAIM 3.3. *After $\mathcal{O}(n \log n)$ rounds two contiguous virtual siblings u_i, u_j are connected by unmarked edges over nodes w with $u_i < w < u_j$ w.h.p. and the graph is weakly connected.*

PROOF. The proof is given by induction over the number of pairs of contiguous virtual siblings v_i, v_j in $[u_i, u_j]$:
Basis: Let u_i, u_j be a pair of contiguous virtual siblings with either $u_j = u_{i-1}$ or $u_i = u_0, u_j = u_m$, such that there is no pair of virtual siblings $v_{i'}, v_{j'}$ in the interval $[u_i, u_j]$. According to rule 6 u_i forms a connection edge to u_j and creates a new connection edge (w_1, u_j) from $w_1 = \max\{w' \in N_u(u_i) : u_i < w' < u_j\}$ to u_j if w_1 exists. Otherwise u_i creates an unmarked backwards edge from u_j to u_i and the claim is fulfilled. Again, based on rule 6, each w_l creates a new connection edge (w_{l+1}, u_j) as long as a $w_{l+1} = \max\{w' \in N_u(w_l) : w_l < w' < u_j\}$ exists. Because there is no pair of virtual siblings $v_{i'}, v_{j'}$ in $[u_i, u_j]$, this is the only command with a true guard and all w_l and w_{l+1} are connected by unmarked edges. If for w_l $w_l + 1$ does not exist, an unmarked backward edge from u_j to w_l is created. Obviously $l \in \mathcal{O}(n \log n)$ w.h.p. Unmarked edges are never converted to ring or connection edges. Thus, either w_{l+1} remains in $N_u(w_l)$ or the edge (w_l, w_{l+1}) is substituted by a path of unmarked edges by the linearization rule.

Inductive step: Let u_i, u_j be defined as above. For all pairs v_i, v_j of contiguous virtual siblings in $[u_i, u_j]$ we know that the induction hypothesis holds. Obviously it takes at most $\mathcal{O}(n \log n)$ rounds until a backwards edge from u_j is created as this is the number of nodes w.h.p.. Let $w_1, \cdots w_l$ be defined as above. We will show that there is a connection between every pair w_l, w_{l+1}. Either $w_{l+1} \in N_u(w_l)$ or $w_{l+1} \in S(w_l)$. In the first case w_l and w_{l+1} are connected with unmarked edges over nodes w with $w_l < w < w_{l+1}$, as w_l and w_{l+1} are neighbors or the edge (w_l, w_{l+1}) is substituted by a path due to linearization. In the second case w_{l+1} is a sibling of w_l and w_l and $w_l + 1$ will be connected via unmarked edges over nodes w with $w_l < w < w_{l+1}$ by the induction hypothesis. Thus in the end all consecutive virtual nodes u_i, u_j are connected by unmarked edges over nodes w $u_i < w < u_j$. □

It might happen that for a real node u new virtual nodes are created in the self-stabilization process. Imagine that after some rounds u is informed about a closest real neighbor that is smaller than its current closest virtual node u_m. Note that this is the only case new virtual nodes are created. All other virtual nodes u_i, $i < m$ do already exist before due to rule 1 and these are eventually connected by unmarked edges by the claim above. Let u and u_m be the existing nodes and $u_{m'}$ the new created closest virtual node with $u < u_{m'} < u_m$. Initially the neighborhood $N_u(u_{m''})$ is empty for all $m' \geq m'' > m$ and so a sequence of unmarked backward edges from u_m to $u_{m'}$ over the $u_{m''}$s is formed by rule 6. The same holds for the pair $u, u_{m'}$ as u is always $u_{m'}$'s closest real neighbor according to rule 3.

CLAIM 3.4. *If a pair of nodes u_i, v_j is weakly connected only by a connection edge $(u_i, v_j) \in E_c$, after $\mathcal{O}(n \log n)$ rounds u_i, v_j are weakly connected by unmarked edges.*

The proof follows from the same arguments as the proof of 3.3.

CLAIM 3.5. *If a pair of nodes u_i, v_j is weakly connected only by a ring edge $(u_i, v_j) \in E_r$, after $\mathcal{O}(n \log n)$ rounds u_i, v_j are weakly connected by unmarked edges.*

PROOF. W.l.o.g we assume $v_j < u_i$, i.e. u_i assumes v_j is missing a left neighbor $< v_j$. From 3.4 we can assume that all nodes are weakly connected by unmarked edges or ring edges. Now there can be four cases: (1) There is a node $w_l \in N(u_i)$ with $w_l < v_j$, (2) there is a node $w_l \in N_r(u_i)$ with $w_l < u_i$ and w_l, u_i are weakly connected by unmarked edges, (3) there is a node $w_l \in N_r(u_i)$

with $w_l < u_i$ and w_l, u_i are not weakly connected by unmarked edges and (4) otherwise.

In case 1 and 2 u_i and v_j are weakly connected by unmarked edges afterwards by the rule 5. In case 3 also by rule 5 one ring edge $(u_i, v_j), (u_i, w_l)$ remains and u_i and v_j are only weakly connected by the remaining ring edge, which will be forwarded to the node $y = \max\{x \in N(u_i)\}$, i.e. $(y, v_j) \in E_r$ in the next round. In case 4 by the same rule the ring edge (u_i, v_j) will be forwarded to the node $y = \max\{x \in N(u_i)\}$. This means that in each round the ring edge is forwarded or u_i, v_j become connected by unmarked edges. If u_i, v_j do not become connected by unmarked edges after $\mathcal{O}(n \log n)$ rounds the connecting ring edge is forwarded to the largest node weakly connected to u by unmarked edges. Note that also the smallest node that is weakly connected to u_i by unmarked edges creates a ring edge. Also this ring edge will be forwarded to the largest and thus after $\mathcal{O}(n \log n)$ rounds case 2 is fulfilled and u_i, v_j are weakly connected by unmarked edges. □

From the Claims 3.3, 3.4 and 3.5 follows Lemma 3.2.

3.1.2 Phase 2: Linearization

After phase 1 each pair of nodes $v, w \in V$ is connected by a not necessarily directed path of unmarked edges. We call such an path a connecting path of v and w. In this phase only the order of a node in $[0, 1)$ is relevant and not its identifier (position). We define the range of an edge to be the difference of the orders of its endpoints and the range of a path to be the difference of the maximal and minimal order of nodes in the path. Considering a pair of consecutive nodes v, w and its connecting path we will show that the range of the path can be decreased to 1 in $\mathcal{O}(n \log n)$ expected rounds, which means that at the end v and w are direct neighbors.

LEMMA 3.6. *After $\mathcal{O}(n \log n)$ rounds a pair of consecutive (in the sorted order) nodes v, w are connected by unmarked edges (v, w) and (w, v) w.h.p..*

We will show this lemma by proving two claims. We firstly look on the node of the smallest order min on the connecting path, assuming $min \neq v$ and $min \neq w$. If min has two outgoing edges $(min, x), (min, y) \in p$, (w.l.o.g. $x < y$), on the path, we will show that the path can be contracted and the range of the path is decreased. In the second claim we will show that after i rounds it takes at most $cn \log n - i$ further rounds until the minimal node on the path has two outgoing edges.

CLAIM 3.7. *Let p be a connecting path of two consecutive nodes v and w, and let min be the minimal node on the path. If min has two outgoing edges $(min, x), (min, y) \in p$, there exists another path p' that connects v, w with a minimal node $min' > min$.*

PROOF. Due to the edges (min, x) and (min, y) either the linearization rule or the overlapping neighborhood rule is applied. See Figure 3. If only the linearization rule is applied, x, y stay connected by a path of unmarked edges with nodes $> min$. Therefore, a new path can be selected connecting v, w with a new node of the smallest order $min' = x > min$. If due to the two edges (min, x) and (min, y) only the overlapping neighborhood rule is applied, (min, x) is forwarded to some min_i and (min, y) to min_j, $min_i \leq min_j$. Lemma 3.2 shows that min_i, min_j are connected by a path of unmarked edges and the path is in the interval of $[min_i, min_j]$. Then also x, y are connected by unmarked edges over nodes u with $min_i < u < min_j$. Therefore a new path connecting v, w via the nodes x, y can be constructed with a new node of minimal order $min' = min_i > min$. If the linearization and the overlapping neighborhood rule are applied, then for x the linearization rule is applied and x is connected with min over min's closest neighbor $z > min$ or x is directly connected as its closest neighbor,then $z = x$. The edge (min, y) is forwarded to min_i and y stays connected with min by Lemma 3.2. This path from min to min_i has to go over min's closest real neighbor. Therefore x, y are connected over z and a path connecting v, w can be constructed with a new minimum $min' \geq z > min$. □

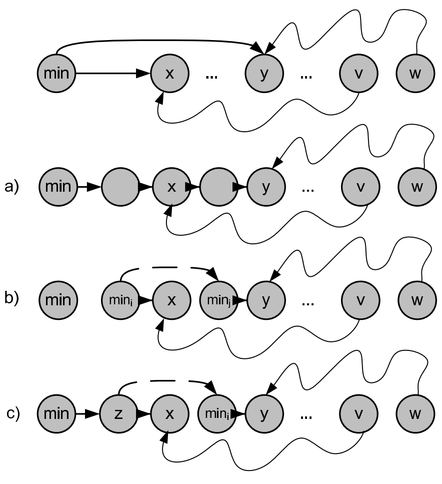

Figure 3: The three cases to increase min on the path from v to w: a)Only the linearization rule, b)only the overlapping neighborhood rule, c) linearization and the overlapping neighborhood rule

To show the second claim we firstly give a construction scheme for the new connecting path. Let p be the old path connecting v, w over min and let p' be the new one. The new connecting path p' is similar to p except that each edge that is forwarded due to the linearization rule can be substituted by a directed path of unmarked edges within the range of the edge due to the linearization. And each edge (u_i, x) that is forwarded to u_j in the overlapping rule can be substituted by a path within the range of the edge between the virtual siblings u_i, u_j, that exists after phase 1 due to Lemma 3.2 and an edge (u_j, x) from the virtual node u_j to x. If an edge (x, y), $x < y$ is mirrored, it is substituted by (y, x). Once min has two outgoing edges on the path the first claim holds and the formerly incoming edges of min are substituted by paths over a new min' according to the proof of Claim 3.7.

CLAIM 3.8. *After i rounds it can take at most $\max\{1, cn \log n - i\}$ rounds till an incoming edge $(x, v) \in p$ with $x > v$ results in an outgoing edge (v, x') for each node $v \in p$.*

PROOF. Proof by induction over the number of rounds i:

Basis(i=0): The longest distance (in number of nodes on a connecting path) between two nodes on p is the total number of nodes $cn \log n$. Thus an edge can be forwarded up to $\mathcal{O}(n \log n)$ times before it is mirrored.

Inductive step($i \rightarrow i + 1$): For all edges on the connecting path p it holds (induction hypothesis) that each incoming edge will be mirrored after at most $cn \log n - i$ rounds. Let $(x, y) \in p$ be such an edge. According to rule 4 the edge is either forwarded or mirrored. An edge that is forwarded, is replaced by a subpath from x to y with all nodes in the interval (x, y) in the connecting path.

This means for all the edges (x',y') on the subpath that the range of (x',y') is less than the range of (x,y), so at most $cn\log n - i - 1$. Thus after at most $cn\log n - (i+1)$ rounds these edges are mirrored. □

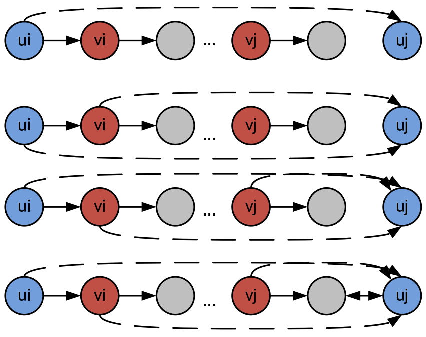

Figure 4: A sequence of connection edges to create a connection between v_i and v_j

Note that outgoing edges are only replaced by outgoing edges constructing the new connecting path p'. Obviously it follows that after i rounds the actual min can be replaced by a $min' > min$ after at most $cn\log n - i$ further rounds. Thus, after $\mathcal{O}(n\log n)$ rounds $min = v$ w.h.p. The same arguments hold for max. Thus, after $\mathcal{O}(n\log n)$ further rounds $max = w$ w.h.p. and v, w are directly connected. Notice that all other rules do not lead to a deletion of unmarked edges, so the described process is valid and two neighbored nodes that are connected will never be disconnected.

3.1.3 Phase 3: Ring

After phase 1 and phase 2 the nodes are ordered in a sorted list. To establish the Re-Chord network the nodes need to form a ring. After phase 2 each node except the minimum and the maximum nodes has a left and right neighbor. Therefore after phase 2 only these two nodes establish marked ring edges.

LEMMA 3.9. *After $\mathcal{O}(n\log n)$ rounds the nodes establish a ring sorted in clockwise order w.h.p..*

PROOF. Let max be the maximum node and min be the minimum node of all nodes. As after phase 2 we already have a sorted list max has no right neighbor and therefore establishes a marked backward edge from the smallest known node. This node is either the minimum node or has a left neighbor, and informs max about it, which will establish a new ring edge from this node to itself. After at most $\mathcal{O}(n\log n)$ rounds max is informed about min w.h.p. and creates the edge (min, max). And analogue min will create the edge (max, min). So after phase 3 all nodes form a sorted ring. □

3.1.4 Phase 4: Closest Real Neighbor

After phase 1, 2 and 3 the nodes are ordered in a sorted ring, i.e. each node has its left and right closest neighbor. However what might be still wrong are the closest real neighbors of some nodes.

LEMMA 3.10. *After $\mathcal{O}(\log n)$ rounds w.h.p. every node knows its closest real neighbors.*

PROOF. Assume that between two neighbored real nodes u, v $u < v$ one node has a missing or wrong closest real neighbor instead of u or v. At least the closest neighbor of u (v) knows v and informs its neighbors (closest neighbor rule). Then after 1 round at least two nodes know u (resp. v) as their closest real neighbor and again inform their neighbors. After $c\log n$ rounds w.h.p. all $c\log n$ (w.h.p.) nodes between u and v are informed about their correct closest real neighbors. □

3.1.5 Phase 5: Finish

After the phases 1-4 the Re-Chord network is finished except some for unnecessary edges. These edges are forwarded up to $\mathcal{O}(n\log n)$ times. Unnecessary edges are edges that might be created during the self-stabilization process but are not part of the desired chord like network. E.g. a edge (u,v) is unnecessary if the edge is unmarked and u and v are no next (real) neighbors. We will show that the length of the longest unnecessary edge decreases with each round.

LEMMA 3.11. *After $\mathcal{O}(n\log n)$ rounds w.h.p. all unnecessary edges are gone.*

PROOF. Let (x,y) be one of the longest unnecessary edges. We know x and y can not be neighbored and so x forwards the edge either to one of its neighbors which exists or to one of its virtual siblings u_i. In the first case the edge is substituted by a directed path $x, x_1, x_2 \cdots, x_n, y$ over the edge to x's closest real neighbor x_1 and a path from the closest real neighbor to y. On this path could be unnecessary edges, but with length $< |(x,y)|$ as $x_i < x_i + 1$ if $y > x$ and $x_i > x_i + 1$ if $y < x$. In the second case the edge (x,y) is substituted by a path $x, x_1, x_2, \cdots, x_l = u_i, y$. The subpath $x, \cdots x_l$ consists of edges $|(x_i, x_i + 1)| = 1$, because every node knows its closest neighbor on the way to x_l. The edge (x_l, y) could be unnecessary, but with a length $< |(x,y)|$.

Assuming that an edge (x', y') with length $|(x,y)|$ is created in this round. Then there can only be three cases. There has been an edge (y', x') which is now mirrored. This can not be the case, because we already showed that every node knows its final closest neighbor. Or it could be that an edge (z', y') is forwarded to x' in the lineariziation rule or the overlapping neighborhood rule. This also can not be the case, because then $|(z', y')| > |(x,y)|$, which would be contradictory to our assumption of the longest unnecessary edge. Therefore after at most $\mathcal{O}(n\log n)$ rounds all unnecessary edges are vanished w.h.p.. □

Proving the Lemmas 3.2, 3.6, 3.9, 3.10 and 3.11 we have shown that at the end of phase 5 a stable Re-Chord structure is reached and as every phase takes at most $\mathcal{O}(n\log n)$ rounds, the complete running time to reach this stable structure is $\mathcal{O}(n\log n)$ rounds.

3.1.6 Stability of Re-Chord

Once a stable Re-Chord structure is reached no further changes will take place. Each node u will perform the stabilization rules. It will not create any new virtual nodes, since in the stable state there is always a node u_m between u and its closest real neighbor. Each u_i, for $0 \le i \le m$ already has one left and one right real neighbor and will create (the already existing edges) to these neighbors. If there does not exist a right/left neighbor (in the case of the largest and smallest node of all nodes) a circle edge is created to the smallest/largest known node, which already existed.Each u_i will sort its neighborhood. At each side u_i has at most two neighbors, as we know. Lets say that at the right/left side u_i/u_{i-1} has edges to v_1/v_2 and r_1/r_2, the closest right/left node and right/left real node. As these neighborhoods are sorted, no overlapping occurs. So after the linearization, in the interval $(id_{u_i}$,$id_{u_{i-1}})$ edges (u_i, v_1), (v_1, r_1), (u_{i-1}, v_2) and (v_2, r_2) are created. These edges obviously existed before. A connection edge is created between the largest neighbor of u_i and u_{i-1}. Also, another connection edge could be present starting u_i, which is a propagated connection edge originally created by another neighborhood. In Re-Chord the same connection edges already existed.Similarly we can show that our network structure is preserved in the case where u_i or u_{i-1} would have only one neighbor , an even more trivial case. We showed the preservation of the stable state for the larger neighbors of u_i. The state is also preserved for its smaller neighbors (if there are any). As u_i is an arbitrary node the above results hold for all u_i, $0 \leq i \leq m$.

4. JOINING OR LEAVING OF A NODE IN THE NETWORK

4.1 Join

We now examine the number of steps needed to successfully integrate a new node to the stable network, which means that the network is again in a stable state. In order to join the network, a peer connects to one peer in the network. Let u be the corresponding new node, which is inserted into the network, i.e. it is connected to an arbitrary real node (of the peer in the network) of the network. We will distinguish two possible cases. Either the node is inserted (connected) to a node smaller than itself, or the opposite. For both cases we will show the following theorem.

THEOREM 4.1. *After at most $\mathcal{O}(\log^2 n)$ rounds, a joining node u is integrated in the Chord network, i.e. every node has stable next and next real neighbors and all virtual nodes are created.*

PROOF. The new node u is initially connected to a real node v. In the first round after the joining u creates its virtual nodes. Then v is the neighbor of one u_i, $v < u_i \leq u$ after performing the overlapping neighborhood rule. As no other (real) node of the network is known to u and its virtual nodes, v is assumed to be u_i's next neighbor and the edge (v, u_i) is created. If $v < u_i$ and u_i's position is between $1/2^{i+1}$ and $1/2^i$ away from v, the edge will be propagated to the virtual node v_j at position $v + 1/2^{i+1}$ and the distance will be at least halved. If there is no such virtual node v_j, u_i falls in the interval between v and the next greater real node. Thus v is its next real neighbor. After the propagation of the edge u_i is not connected to a real node $< u_i$, but a virtual node $v_j < u_i$. Then a real node $v_j < v' < u_i$ will be found in one round if such a node exists. If v_j's next greater real neighbor has a smaller id than u_i the edge (v', u_i) will be created by the linearization rule. If the next greater real neighbor of v_j is greater than u_i, v_j and u_i fall in the same interval of real nodes and v_j next real neighbor is u_i's next real neighbor. Thus in every second round the distance to u_i is halved and it takes at most $\mathcal{O}(\log n)$ rounds until u_i is connected to one of its next real neighbors. From that point on the procedure is trivial. In $\mathcal{O}(\log n)$ rounds u_i will be connected to its stable-state neighbors, since we showed that between two consecutive real nodes there are no more than $\mathcal{O}(\log n)$ virtual nodes w.h.p.. If $v < u_i$, v is the right neighbor of one u_i, $v > u_i \geq u$. If there is another $u_{i-1} > u_i$ a connection edge (v, u_{i-1}) will be created to connect u_{i-1}, u_i. With the same arguments as above u_{i-1} will be connected to its stable-state neighbors after $\mathcal{O}(\log n)$ rounds. But if there is no $u_{i-1} > u_i$, v will not be the left neighbor of any of the u_is. In this case the smallest virtual node of u (or u itself), lets call this node y, will create a circle edge from the largest known node to u to y. The largest known node so far is v so a circle edge (v, y) is created. This edge is also propagated according to the circle edge rules, and for the propagation procedure the same arguments hold as for the former case and y will reach its stable-state neighbors in $\mathcal{O}(\log n)$ steps.

As soon as the first virtual node is fully integrated in the ring structure by case 1 or 2 it knows its next real neighbors and will connect them by a connection edge with its next virtual sibling u_j. This will also be integrated following the argument of case 1 or case 2. So, until the last virtual sibling of u is integrated, $\mathcal{O}(\log n)\Theta(\log(n)) = \mathcal{O}((\log n)^2)$ rounds will be needed. After the integration of u and its virtual nodes the routing and search mechanisms of Chord can be applied again, because all the information of the old nodes of the former ring are updated during the joining process. An old real node v creates only a new smallest virtual node v_m if the joining node u is its new next real neighbor, Thus this new virtual node v_m is integrated in at most $\mathcal{O}(\log n)$ rounds. Also all virtual nodes in the corresponding interval are informed about their new next real neighbor u in at most $\mathcal{O}(\log n)$ rounds after u is connected to its next neighbors. Note that there still might be unnecessary edges created during the joining process, that will be eliminated after at most $\mathcal{O}(n \log n)$ rounds. □

4.2 Leave

This case is simpler than the insertion. A node can either leave the network, or a fault can occur and the node, as well as its connections, fail. When a node leaves the network, it and all of its virtual nodes will be deleted. Before a node is deleted it informs its neighbors about each other and so the ring structure is maintained. When a node fails, the network is also able to recover to its ring structure.

THEOREM 4.2. *After at most $\mathcal{O}(\log n)$ rounds the Chord network is stabilized again after the leaving or failure of a node.*

PROOF. If a node fails it can not inform the neighbored nodes about its failure. When a virtual node fails a "gap" between two consecutive nodes exists that is filled with an edge at most after 2 rounds. These nodes realize their next neighbor is now their next real neighbor, an edge from that real neighbor to the node is created and after at most $\mathcal{O}(\log n)$ rounds the desired edge is created due to the linearization rule. When a real neighbor fails, a similar gap is created, but now the next real neighbor of the neighbored nodes is missing. But the nodes at the gap create new connection or ring edges according to rule 5 and 6, which will close the gap after at most $\mathcal{O}(\log n)$ rounds with the same arguments as for the joining process, as the rest of the ring is maintained. □

5. SIMULATIONS

As a simulation environment for our algorithm we use Matlab 7.8.0. We simulate a random undirected weakly connected graph. Each vertex represents a node of the chord network and has a real number (id) assigned to it, which is chosen uniformly at random from (0,1). This number also indicates the position of the node in the chord network circle. The vertices present at initialization represent the real nodes. The self-stabilization rules are applied repeatedly to the nodes of the graph. After some steps the graph has reached the desired stable state of the chord network.

The metrics that are measured are the number of steps it takes for the network to stabilize, the number of edges that exist at the stabilization state (normal edges as well as connection edges) and the total number of nodes that exist in the network (the real nodes of the initialized state, as well as the virtual nodes produced). By number of steps we mean the number of times a real node applies to itself (and to its virtual nodes) the self-stabilization rules. Note that nodes work in parallel.

The simulations are run for various numbers of (real) nodes: 5, 15, 25, 35, 45, 65, 85, 105. For each of these scenarios we run the simulation for 30 different graphs and compute then the mean value of the values of the metrics we get.

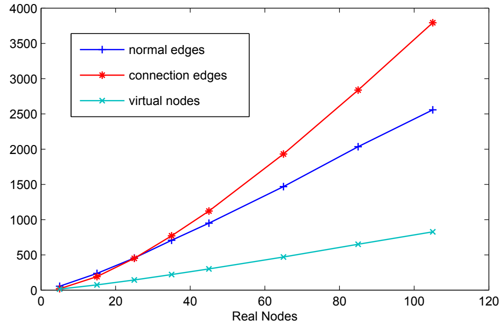

Figure 5: Edges and nodes measured from various simulation runs of the algorithm

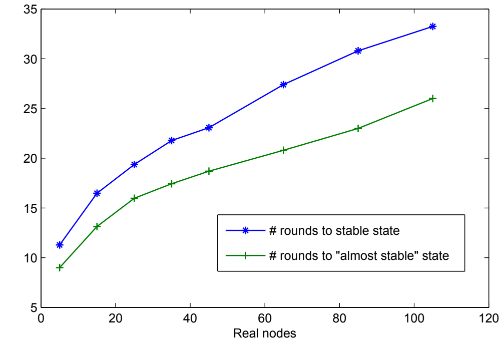

Figure 6: Number of steps needed to reach the stable state and "Almost stable" state

As we derived from our simulations, the network indeed stabilizes after a number of rounds, and the stabilization state is indeed the chord network state. This shows that our algorithm works correctly.

We also can see from Fig. 6 that the number of steps needed in order to reach the stabilization state is relatively small. In particular, the steps needed at low numbers of nodes are from 10 to 25 (for 30 nodes) and don't get much more for higher numbers. They seem to increase sublinear, or at most linear. Here, a gap between the experimental results and the results from the analysis seems to exist, where we showed that the convergence to the Chord structure takes $\mathcal{O}(n \log n)$, whereas the simulations show that the steps needed are (at most) linear. This implies that our upper bound may not be a tight one. We can also see in the figure that the network converges relatively early to an "almost stable" state, before it gets its final stable state. The "almost stable" state describes a network, where all the desired edges of the Re-Chord network exist, but also some extra edges exist. In Fig. 5 we consider the number of edges and nodes. In particular, we measure the amount of connection edges, which are the ones created due to rule 6 of our algorithm. By normal edges we mean all the other edges that exist and are created due to the algorithm except the connection edges. We can see a remarkable smoothness in Figure 5, which also indicates the small variation of the metrics that was observed during the experiments. The normal edges seem to increase a bit faster than linear, as expected. It is notable that the connection edges increase faster than the normal edges, as the number of real nodes gets higher. This is no surprise, as described in Section 2.2. The curve seems to be almost identical to a $cn(\log n)^2$ curve that follows from the theory. If we add the edges and take into account the number of total edges, we can see that it increases with a rate compared to to the total number of nodes, which seems to support the theory, as described in Section 2.2. The virtual nodes increase at least linear, which supports the theoretical result, as there are $\mathcal{O}(n \log n)$ virtual nodes. By number of total edges we mean the sum of the connection and normal edges, and by number of total nodes we mean the sum of the real and virtual nodes.

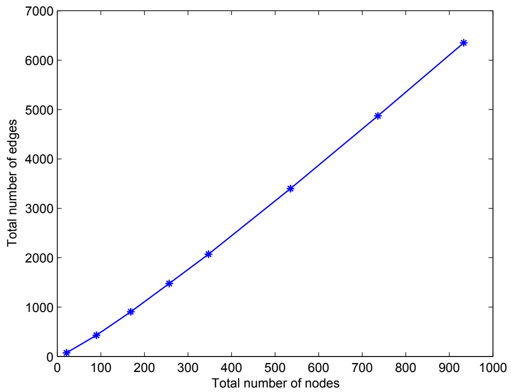

Figure 7: The total number of edges to the total number of nodes in the final graph

6. CONCLUSIONS

We showed that it is possible for a network to reach a Chord-network state in a distributed manner, i.e. using only local actions, from any arbitrary structure and remain at that state. In fact these local actions are a set of rules, based on the principle of the linearization technique, which basically consists of sorting all neighbors of each node into a line. We extended this technique by expanding the rules in a way, in order to deal with the problems occurred by trying to self-stabilize the graph into a Chord state. This convergence was shown through rigorous analysis, but also by the simulation experiments we conducted. We also saw through simulations that this self-stabilization happens in a relatively small number of steps and by creating not too many edges. By analyzing the algorithm we proved that this convergence to the stable Chord structure takes always at most $cn \log n$ steps, where n is the size of the network, i.e. the number of peers. We also showed that once a network is in the stable state, and a peer joins or leaves the network, the network will recover to the stable state at most at $\mathcal{O}((\log n)^2)$ steps at the case of joining and at most $\mathcal{O}(\log n)$ at the case of leaving. It would be interesting to further investigate if there could be even more efficient rules that lead to self-stabilization, or study other types of graphs that could be formed in a self-stabilization process, from an arbitrary weakly connected network.

7. REFERENCES

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

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

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

[4] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting. In *SPAA*, pages 170–179, 2004.

[5] J. Brzezinski, M. Szychowiak, and D. Wawrzyniak. Self-stabilization in distributed systems - a short survey, 2000.

[6] T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list. In *SSS*, pages 124–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] C. Cramer and T. Fuhrmann. Self-stabilizing ring networks on connected graphs. Technical report, University of Karlsruhe (TH), Technical Report 2005-5, 2005.

[8] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, November 1974.

[9] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[10] 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.

[11] 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*, pages 9–9, 2003.

[12] T. Herman. Self-stabilization bibliography: Access guide, December 2002.

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

[14] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid. A self-stabilizing and local delaunay graph construction. In *Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 771–780. Springer Berlin / Heidelberg, 2009.

[15] J. Jiang, R. Pan, C. Liang, and W. Wang. Bichord: An improved approach for lookup routing in chord. In *ADBIS*, pages 338–348, 2005.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997. ACM.

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

[18] B. Leong, B. Liskov, and E. D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. In *ICON*, pages 1243–1259, 2004.

[19] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC*, pages 183–192, 2002. ACM.

[20] V. A. Mesaros, B. Carton, P. V. Roy, and P. S. Barbe. S-chord: Using symmetry to improve lookup efficiency in chord. In *PDPTA03*, pages 23–26, 2003.

[21] A. Montresor, M. Jelasity, and Ö. Babaoglu. Chord on demand. In *Peer-to-Peer Computing*, pages 87–94, 2005.

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

[23] M. Onus, A. W. Richa, and C. Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *ALENEX*, 2007.

[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 '01*, pages 329–350, 2001. Springer-Verlag.

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

[27] A. Shaker and D. S. Reeves. Self-stabilizing structured ring topology p2p systems. In *Peer-to-Peer Computing*, pages 39–46, 2005.

[28] I. Stoica, R. Morris, D. Liben-nowell, D. Karger, M. Frans, K. F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[29] J. Wang and Z. Yu. A new variation of chord with novel improvement on lookup locality. In *GCA*, pages 18–24, 2006.

[30] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41 – 53, Jan. 2004.