Ca-Re-Chord: A Churn Resistant Self-stabilizing Chord Overlay Network

Markus Benter*, Mohammad Divband, Sebastian Kniesburges†, Andreas Koutsopoulos† and Kalman Graffi‡
*Algorithms and Complexity Group and †Theory of Distributed Systems Group, University of Paderborn, Germany
‡Technology of Social Networks Group, University of Düsseldorf, Germany
Email: graffi@cs.uni-duesseldorf.de

***Abstract*—Self-stabilization is the property of a system to transfer itself regardless of the initial state into a legitimate state. Chord as a simple, decentralized and scalable distributed hash table is an ideal showcase to introduce self-stabilization for p2p overlays. In this paper, we present Re-Chord, a self-stabilizing version of Chord. We show, that the stabilization process is functional, but prone to strong churn. For that, we present Ca-Re-Chord, a churn resistant version of Re-Chord, that allows the creation of a useful DHT in any kind of graph regardless of the initial state. Simulation results attest the churn resistance and good performance of Ca-Re-Chord.**

I. Introduction

Peer-to-peer (p2p) networks have a long history in networking research. Their main characteristic is the self-organizing approach to interact decentralized and to provide powerful networking functions. Since their advent in the early 2000s, their application range extended from filesharing, multimedia streaming to online social networks [7], sometimes serving several Millions of users using only the resources contributed by the user devices. According to [12], p2p becomes a business relevant paradigm. One main advantage of the p2p paradigm over centralized approaches is the robustness of the network against failures of individual nodes. While the central node in a centralized network is a single point of failure, in a typical p2p overlay churn is expected and handled. Still, many p2p overlays require a strict bootstrapping procedure, e.g. using a bootstrap node or a churn-safe join procedure.

However, in the worst case the network starts from a totally randomized graph. From this point, the desired p2p overlay should recover. Even more, a very challenging but desirable property for systems was defined by Dijkstra in [5]. Dijkstra calls "a system *self-stabilizing* if regardless of the initial state, the system is guaranteed to find itself in a legitimate state after a finite number of moves". Applying the idea to Chord [18], the most cited p2p overlay, implies that a Chord network should be created within a finite number of local actions starting from an arbitrary network. Thus churn, network initialization and robustness issues would be no longer an issue. The application areas are numerous, ranging from robust large-scale p2p-based social networks to self-organizing, p2p connected robot swarms in hostile environments. Chord offers routing in $O(\log N)$ and is due to its history and many extensions an ideal candidate to show the approach of self-stabilization. In this paper we present 1) our approach to turn Chord self-stabilizing (Reactive Chord) and 2) a practical extension, churn-aware Re-Chord (Ca-Re-Chord), which accelerates and protects the self-stabilization process. With the presented approach Chord can be created from arbitrary graphs using a robust transformation process.

The paper is structured as follows. In Section II we motivate the challenge of self-stabilization and present related work on creating specific network structures from arbitrary networks. In Section III we present Reactive Chord (Re-Chord), which is a self-stabilizing network able to form itself from an arbitrary graph. It contains the Chord topology as a subset and implements the same lookup function. This section displays in detail the stabilization rules and sketches the proofs for self-stabilization. While Re-Chord is able to turn an arbitrary network in Chord, its stabilization process itself is prone to churn. Thus, in Section IV we present a churn-aware Re-Chord (Ca-Re-Chord) that inherits both the characteristics of Re-Chord but comes also with churn-aware additional extensions. Section V evaluates Chord, Re-Chord and Ca-Re-Chord and shows the resistance of Ca-Re-Chord against churn, which is comparable to the resistance of Chord. The paper concludes in Section VI. We demonstrate that Ca-Re-Chore is able to deal with Churn and performs effectively under high Chun.

II. Related Work

In 1974, self-stabilization was introduced into the field of distributed computing by Dijkstra in [5]. With the upcome of p2p networks like Chord [18] and further overlays, the question was restated for these. In [4], Clouser et al. describe Tiara, a self-stabilizing deterministic skip list using a self-stabilizing peer-to-peer network maintenance algorithm and supporting logarithmic searches and updates. The authors of Tiara briefly present Chord as a possible extension of Tiara to a ring structure but do not elaborate it. In [1], the authors suggest self-stabilization as an general approach to make overlay networks fault tolerance. Janson et al. present 3nuts in [9], which is a self-stabilizing p2p network supporting range queries and adapting the overlay structure to the underlying physical network. Bern et al. describe in [2] a (theoretical) transitive closure framework, for the self-stabilizing construction of any overlay networks. Its applications to practical

This work was partially supported by the FET project MULTIPLEX 317532 and the German Research Foundation (DFG) within the Collaborative Research Centre „On-The-Fly Computing" (SFB 901)

overlays and networks remains open. Onus et al. [15] presented a linearization technique to transform an arbitrary connected network into a sorted list. Later, Richa et al. [16] tried to extend this technique to present the linearized De Bruijn network, which is based on a discretization of a continuous variant of the classical De Bruijn. The linearized De Bruijn is self-stabilizing and can recover from any case in which the topology is still weakly connected. The algorithm tries to be fault tolerance by maintaining constant node degree. In [10], we motivate self-stabilization for p2p overlays and prove the convergence to a stable and valid state for Chord.

In this paper, we present and extend Re-Chord with practical protocol elements, to both (self-)stabilize the operation of the overlay as well as the stabilization process. In contrast to related and previous work, we enable the prominent overlay Chord with self-stabilization properties and show in the evaluation the stable behavior of Ca-Re-Chord under churn.

III. Re-Chord: Self-stabilizing Chord

According to [3], a system is called *self-stabilizing* if and only if *"regardless of current system's state, the system will converge to a legal state in a finite number of steps"*. In this section, we introduce Re-Chord, which achieves this property in the following way: If Re-Chord is in a stable state then a Chord [18] network is a subgraph of the Re-Chord network. Please note, we modeled self-stabilization characteristics of Chord in [10] and gave proofs on its self-stabilization property.

A. Chord

Chord [18], [19] is the most cited p2p overlay today and the basis for our self-stabilizing Re-Chord. Chord is a one dimensional circle space with clockwise directional links. The main functionality of the Chord protocol is to look up a node for a given key. For that, both nodes and objects have a 160 bit identifier. In the one dimensional ring, every node knows its identifier and maintains a link to its predecessor and successor in the ring. Each node has the responsibly of the keys (which we call k), within predecessor's id $<$ k $\leq$ current node's id. Thus, the current node is the $successor(k)$ of the object id k. In order to accelerate routing, i.e. the look up of a node responsible for an object id, each node has a routing table of size $O(logN)$ containing identifiers (fingers) to nodes in exponentially growing distances. Routing is done by forwarding a lookup request to the smallest finger not surpassing the queried object id. Thus, with every hop towards the responsible peer the distance is (at least) halved. In the final step, the predecessor of the responsible node passes the lookup request to the destination node.

B. Creation of Re-Chord

For simplicity and generality, we assign each node u an identifier in the $[0, 1)$-interval (instead of the 160-bit identifier as defined in Chord). If we talk about u then we often mean its identifier if the context is clear. For the stabilization process, we use an extended *local linearization* as first described in [15]. This is a self-stabilizing procedure. This linearization is able to order the nodes ascending according to its identifiers.

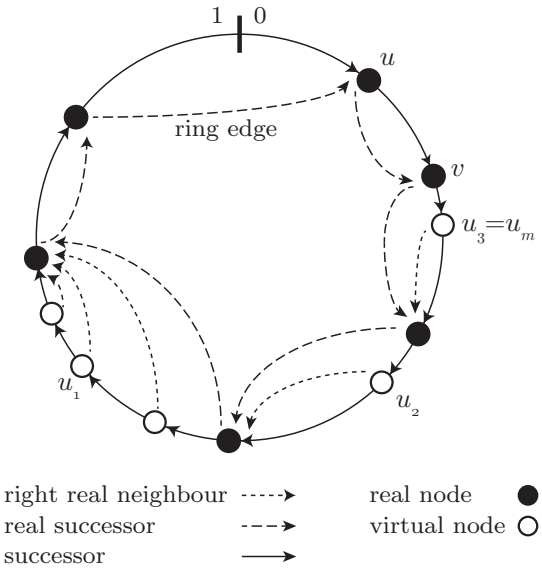


Fig. 1. Re-Chord: Edges and Virtual Nodes in Stable State.

This already forms the Chord ring but however, the fingers cannot be expressed as a line. Therefore, we redefine the fingers of Chord as so called *virtual nodes*: Each node u has virtual nodes $u_i = u + \frac{1}{2^i} \mod 1$. Furthermore, we define $u_0 := u$. This virtual nodes simply mean that a node joins the network multiple times with different identifiers. Now the virtual nodes are sorted together with the real nodes using the *local linearization*. Thus, they can be used as fingers. Now the *stable state* of a Re-Chord network is defined as follows:

- Each *node* (either virtual or real) has an edge to its following and preceding node (either virtual or real). This nodes are called *successor* and *predecessor*.
- Each *real node* has an edge to its following and preceding real node (called *real successor* and *real predecessor*).
- Each *virtual node* has an edge to its following and preceding real node (called *right real neighbor* and *left real neighbor*)

An example can be found in Fig. 1. For simplicity, only the edges to following nodes are represented. The same can be done for preceding nodes. The Chord sub-graph is now a subset of the edges described above combined with using virtual nodes:

- The edges of real nodes to its following real nodes define the Chord-circle (*real successor* in Fig. 1)
- The virtual nodes can be used as fingers: If a node u wants to use a finger, it sends the message using the corresponding virtual node u_i. More precisely, the edge from the virtual node u_i to its right real neighbor (see Fig. 1, *right real neighbor* edges).

All other edges are not relevant for routing but necessary for the stabilization process (see Section III-C). It can be observed that some of the virtual nodes are not necessary: Consider a real node u and one of its virtual nodes u_i. If there is no real node between u and a u_i, then u_i's real right neighbor is u's successor. Because u has already a direct edge to this node,

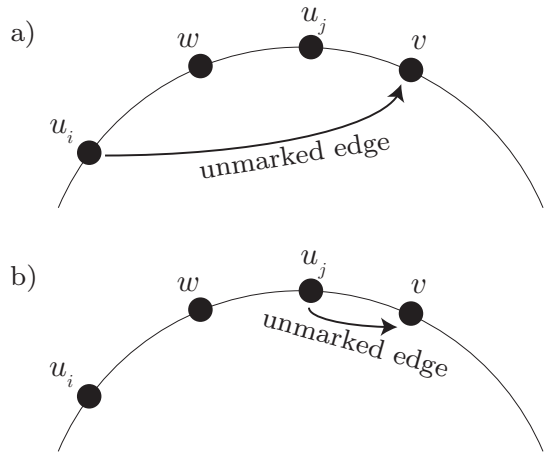


Fig. 2. Overlapping Neighborhood

u_i is useless and can be removed (or not created). For later purpose, we define the first index a virtual node is created as m. With other words, u_m is the virtual node with the smallest distance to u. An example can be found in Fig. 1: As it can be observed, the real node u has three virtual nodes $u_1 = u + \frac{1}{2}$, $u_2 = u + \frac{1}{4}$ and $u_3 = u + \frac{1}{8}$ but there is no virtual node between u and its following real successor v. Therefore, $m = 3$ holds. For the stabilization procedure, we have to maintain three types of edges:

- *Unmarked edges* E_u: All normal edges, e.g. all edges shown in Fig. 1 (with exception of the ring edge).
- *Ring edges* E_r: The edge that closes the ring between the nodes left and right of the 0-1 transition (see Fig. 1). Possibly more than one edge before stable state is obtained.
- *Connection edges* E_c: A special type of edges needed for stabilization process only (see section III-C).

We further define the following notations:

- $N_u(u_i)$ is the *unmarked neighborhood* of u_i.
- $N_r(u_i)$ is the neighborhood according to the outgoing *ring edges* of u_i.
- $N_c(u_i)$ is the neighborhood according to the outgoing *connection edges* of u_i.
- $S(u_i) = \{u_0, u_1, ..., u_m\}$ is the set of *siblings* of u_i.
- $N(u_i) = S(u_i) \cup \bigcup_{0 \leq j \leq m} N_u(u_j)$ is the *neighborhood* of u_i and contains all unmarked edges of siblings of u_i plus the siblings itself.

C. Stabilization Rules

The stabilization rules are applied locally in each node periodically in order to have as a global effect the self-stabilization of the network towards a Chord network. For the execution of the stabilization rules we consider a round based communication model. We define the following two assignments in this model:

- $x := y$ assigns y directly to x (in the same round)
- $x \leftarrow y$ assigns y to x in the next round

The delayed assignment ($\leftarrow$) indicates that communication is needed, i.e. the state of a neighbor is changed. Now, we will present six stabilization rules. Each rule will be first described informal and then a formal pseudo-code is given.

1) Virtual Nodes: As already described above, u_m is the first virtual node succeeding u. Therefore, the rule is the following: Create all virtual nodes u_i with $i \leq m$ and delete all virtual nodes with $i > m$. In order to not lose edges, all neighbors of deleted nodes are transferred to u_m.

```
CREATEVIRTUALNODES(u)
for all i ≤ m do
  if u_i ∉ S(u) then
    S(u) := S(u) ∪ {u_i}
  end if
end for

DELETEVIRTUALNODES(u)
for all u_i ∈ S(u) do
  if i > m then
    S(u) := S(u)\{u_i}
    N_u(u_m) := N_u(u_m) ∪ N_u(u_i) ∪ N_r(u_i) ∪ N_c(u_i)
  end if
end for
```

2) Overlapping Neighborhood: Consider two siblings u_i and u_j. If there is an edge from u_i to v and u_j is closer to v than u_i (see Fig. 2a) the edge should be transferred to u_j (see Fig. 2b). Note: There are no communication costs because u_i and u_j are the same real node.

```
CHECKOVERLAPPINGNEIGHBORHOOD(u)
for all u_i ∈ S(u) do
  for all u_j ∈ S(u) do
    for all w ∈ N_u(u_i) do
      if w < u_j < u_i ∨ w > u_j > u_i then
        N_u(u_j) := N_u(u_j)
        N_u(u_i) := N_u(u_i)\{w}
      end if
    end for
  end for
end for
```

3) Closest Real Neighbor: For each u_i the *real right neighbor* $r_r(u_i)$ is updated to the real node $w \in N(u_i)$ which has a higher id than u_i but is as close as possible to u_i. The *real left neighbor* $r_l(u_i)$ is calculated accordingly. Furthermore, each node in the interval $[r_l(u_i), r_r(u_i)]$ is informed about the new found neighbor for updating its real neighbors.

An illustration of the worst-case of finding the right real neighbors is presented in Fig. 3. Lets assume we have k contiguous virtual nodes. Now $u^1_{i_1}$ updates its real right neighbor to its successor (Fig. 3a). Then $u^1_{i_1}$ announces the new real neighbor to all nodes between its left real neighbor and right real neighbor. This interval is $[u^1, v]$ (the real node of $u^1_{i_1}$ and the successor of $u^1_{i_1}$) in the worst case. Therefore, (at least) $u^2_{i_2}$ receives an announcement and creates an edge to v (Fig. 3b). The same is done for $u^3_{i_3}$ up to $u^k_{i_k}$. As it can be observed, the worst case time for updating right real neighbors is the number of contiguous virtual nodes k. However, this may be faster if additional edges are available.

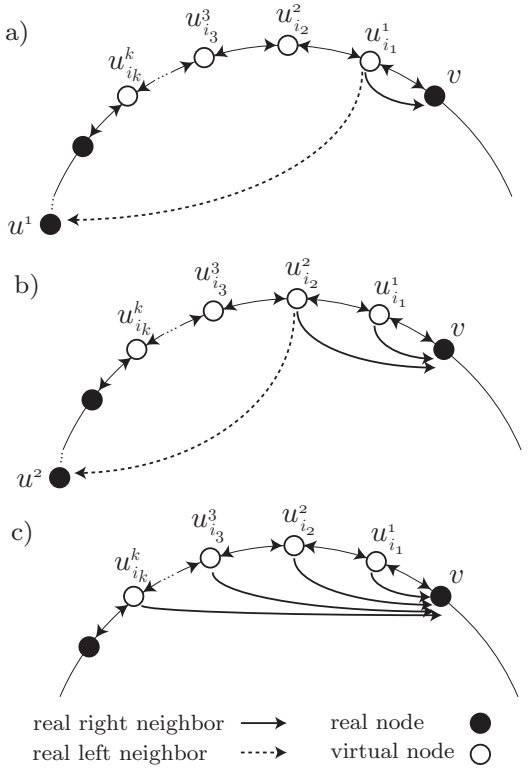


Fig. 3. Find Real Right Neighbors, Worst-Case

UPDATEREALNEIGHBORS(u)
for all $u_i \in S(u)$ **do**
  $r_r(u_i) := \max(w \in N(u_i) \mid w \text{ is real node } \wedge w < u_i)$
  $r_l(u_i) := \min(w \in N(u_i) \mid w \text{ is real node } \wedge w > u_i)$
  $N_u(u_i) := N_u(u_i) \cup \{r_r(u_i)\} \cup \{r_l(u_i)\}$
  for all $v \in N(u) \wedge v \in [r_l(u_i), r_r(u_i)]$ **do**
    $N_u(v) \leftarrow N_u(v) \cup \{r_r(u_i)\} \cup \{r_l(u_i)\}$
  end for
end for

LINEARIZE(u)
for all $u_i \in S(u)$ **do**
  // Linearize left
  Sort $v^j \in N_u(u_i) \wedge v^j < u_i$ as $v^1 < v^2 < ... < v^k$
  for $j := 1$ **to** $k-1$ **do**
    $N_u(v^{j+1}) \leftarrow N_u(v^{j+1}) \cup \{v^j\}$
    $N_u(u_i) := N_u(u_i) \backslash \{v^j\}$
  end for
  // Linearize right
  Sort $w^j \in N_u(u_i) \wedge w^j > u_i$ as $w^1 > w^2 > ... > w^l$
  for $j := 1$ **to** $l-1$ **do**
    $N_u(w^{j+1}) \leftarrow N_u(w^{j+1}) \cup \{w^j\}$
    $N_u(u_i) := N_u(u_i) \backslash \{w^j\}$
  end for
  // Mirroring
  for all $v \in N_u(u_i)$ **do**
    $N_u(v) \leftarrow N_u(v) \cup N_u(u_i)$
    $N_u(u_i) := N_u(u_i) \cup \{r_l(u_i)\} \cup \{r_r(u_i)\}$
  end for
end for

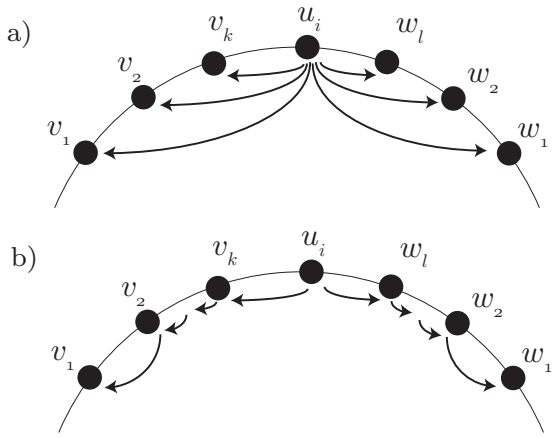


Fig. 4. Linearization Applied on u_i

4) Linearization: The linearization routine forwards all unmarked edges of u_i with exception of the two closest ones (with higher respectively lower identifier than u_i). An edge (u_i, v) is *forwarded* to (w, v) where w is the node with the closest identifier to v. An illustration can be found in Fig. 4, a) before linearization and b) after linearization applied on u_i. In [15], it has been shown that this approach (algorithm *Pure Linearization*) converges to a undirected line in $O(n)$ steps.

Furthermore, so called backward edges are created: If an edge (u_i, v) is still available after the *forwarding* process, a back edge (v, u_i) is created. We call this *mirroring*. Additionally, right real neighbors $r_r(u_i)$ and left real neighbors $r_l(u_i)$ are preserved by adding them to the unmarked edge set after the forwarding procedure.

5) Ring Edge: The linearization process described above is only able to create a list but the edge between the node with the lowest identifier u_{min} and the highest identifier u_{max} is missing. Therefore, a node creates a ring edge whenever it thinks it is the node with the highest or lowest identifier in the network according to its local knowledge. Every time a node u has no neighbor with *higher* identifier, it creates a ring edge (v, u) where $u \in N(u)$ is the neighbor with the lowest id. If v knows a node w with lower id than itself it forwards the ring edge: (v, u) is removed and (w, u) is created. w forwards again if possible and so on. Forwarding may happen if the network is not yet stable. Every time a node u has no neighbor with *lower* identifier, the procedure works accordingly.

6) Connection Edges: The connection edges are able to close gaps between contiguous virtual nodes. Therefore, a node u_i creates a connection edge to node u_{i-1}. If a node u has an outgoing connection edge (u, v) it is replaced by (w, v) as long as a $w \in N_u(u_i) \cup S(u_i)$ with a higher identifier than u but lower identifier than v exists. If such a w does not exist, an unmarked edge (v, u) is created and the connection edge (u, v) is removed.

An example is shown in Fig. 5. Assume that u_{i-1} does not know any node with id in the interval $[u_i, u_{i-1}]$. First, a connection edge between u_i and u_{i-1} is created (Fig. 5a). Then, the edge is forwarded to the node with highest id v but lower than u_{i-1} (Fig. 5b). Then, the edge is again forwarded (Fig. 5c). x is not considered as $x < p_{i-1}$ does not hold.

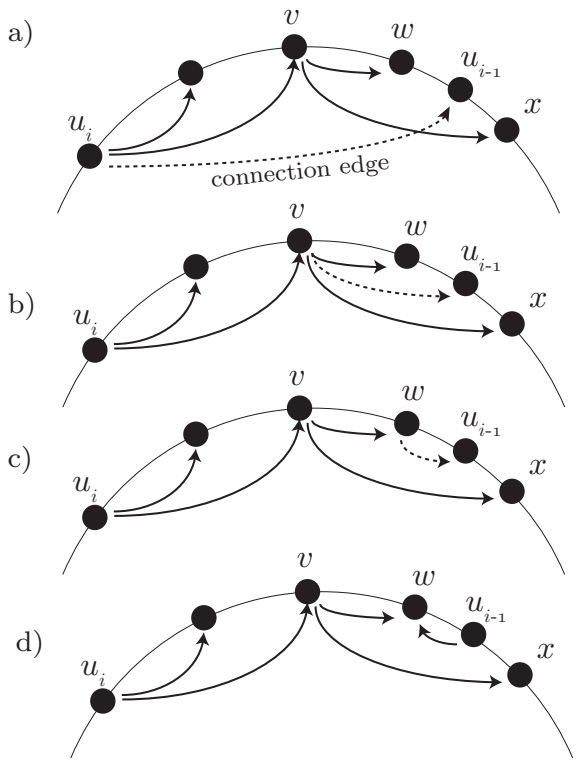


Fig. 5. Connection Edge Added between u_i and u_{i-1}

MANAGERINGEDGES(u)
for all $u_i \in S(u)$ **do**
  // Create ring edges
  if $\nexists v \in N(u_i) : v < u_i$ **then**
    $v_{max} := \max(N(u))$
    $N_r(v_{max}) \leftarrow N_r(v_{max}) \cup \{u_i\}$
  end if
  if $\nexists v \in N(u_i) : v > u_i$ **then**
    $v_{min} := \min(N(u))$
    $N_r(v_{min}) \leftarrow N_r(v_{min}) \cup \{u_i\}$
  end if
  // Forward ring edges
  for all $v \in N_r(u_i)$ **do**
    if $v < p$ **and** $\exists w \in N(u_i) : w > u_i$ **then**
      $v_{max} := max(N(p))$
      $N_r(v_{max}) \leftarrow N_r(v_{max}) \cup \{w\}$
      $N_r(u_i) := N_r(u_i) \backslash \{w\}$
    end if
  end for
  for all $v \in N_r(u_i)$ **do**
    if $v > p$ **and** $\exists w \in N(u_i) : w < u_i$ **then**
      $v_{min} := min(N(p))$
      $N_r(v_{max}) \leftarrow N_r(v_{max}) \cup \{w\}$
      $N_r(u_i) := N_r(u_i) \backslash \{w\}$
    end if
  end for
end for

At the end, an unmarked edge (w, u_{i-1}) is created because the edge cannot be forwarded anymore (Fig. 5d). Note that u_i has found a new predecessor in the interval $[u_i, u_{i-1}]$.

MANAGECONNECTIONEDGES(u)
// Create connection edge
for $i := 2$ **to** m **do**
  $N_c(u_i) := N_c(u_i) \cup \{u_{i-1}\}$
end for
// Forwarding
for all $u_i \in S(u)$ **do**
  for all $v \in N_c(u_i)$ **do**
    $v_{max} := \max\{x \in N_u(u_i) \cup S(u_i) | x < v\}$
    if $v_{max} \neq u_i$ **then**
      // Forward connection edge
      $N_c(v_{max}) \leftarrow N_c(v_{max}) \cup \{v\}$
      $N_c(u_i) := N_c(u_i) \backslash \{v\}$
    else
      // Create unmarked edge
      $N_u(v) \leftarrow N_u(v) \cup \{u_i\}$
      $N_c(u_i) := N_c(u_i) \backslash \{v\}$
    end if
  end for
end for

D. Analysis

Having these six rules, Re-Chord is self-stabilizing. In [10], we give a model for Re-Chord and prove its properties. Here, the most important theorems of Re-Chord are summarized.

Theorem 1: After $O(\log n)$ rounds w.h.p every node knows its closest real neighbors.

Proof: As part of the *worst case example* in paragraph III-C3 it has already be shown that the *closest real neighbors* can be found in O(*number of contiguous virtual nodes*). According to the definition of virtual nodes, each real node has $O(\log(n))$ virtual nodes w.h.p. Thus, we have $O(\log(n))$ virtual nodes between two preceding real nodes x and y w.h.p. It follows that each virtual node can find its *right real neighbor* and *left real neighbor* in $O(\log n)$ rounds w.h.p. ■

Theorem 2: After $O(n \log n)$ rounds the nodes establish a ring sorted in clockwise order w.h.p.

Theorem 3: 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 sub-graph.

Theorem 4: After at most $O(log^2 n)$ rounds, a joining node u is integrated in the Chord network, i.e. every node has stable successor, real successors and real neighbors and all virtual nodes are created.

Theorem 5: After at most $O(\log n)$ rounds the Chord network is stabilized again after the leaving or failure of a node.

IV. CA-RE-CHORD: CHURN AWARE RE-CHORD

While Re-Chord is able to self-stabilize from an arbitrary weakly connected network to Chord, it is prone to dynamics during the stabilization time. Evaluations show in Section V that Re-Chord is not churn resistant, i.e. the number of succeeded queries are low even with reasonable amount of joining and leaving nodes. According to our analysis, the main reason is that the stabilization process is too slow. Even if $O(\log n)$ message transfers seem to be acceptable

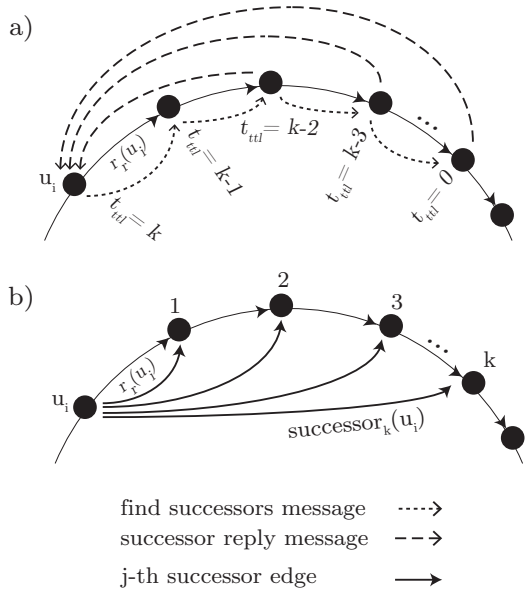


Fig. 6. How To Find Successors

for stabilizing after a single node leaves (Theorem 5) and $O(\log^2 n)$ after a single node joins (Theorem 4), further joins and leaves may happen with a contiguous churn model before stable state is obtained and thus run-time can increase up to $O(n \log n)$ (Theorem 3). Therefore, we extend Re-Chord to *Ca-Re-Chord*, a churn aware variant of Re-Chord. Ca-Re-Chord stores up to a constant number k of successors for quick-fixing gaps in the Chord ring.

A. Communication Model

The following approach depends on the communication model details, thus, we first introduce the *discrete-event based model*. Each node can schedule *operations* that are either executed immediately or after a specified time. If a message has been sent to a node, delivery can not be guaranteed. Typically, an *acknowledge message* indicates that the message has been arrived. If no acknowledge message arrives after a certain time, message loss or node failure can be assumed.

B. Datastructure

Each real and virtual node u_i has a datastructure for storing the next k *real* nodes. A *real* node stores the next k *real successors*, a *virtual* one for the next k *right real neighbors*. The data can be accessed via $successor_j(u_i)$ with $1 \leq j \leq k$. The datastructure is updated by sending out messages periodically every T_{update} time steps. Each message has a certain *time to live* t_{ttl}, initialized with k by the initiating node and decremented by one after every hop. Every node v which receives such a message sends back a message to the initiating node, including the information that the node is the j-th successor by calculating $j := k - t_{ttl}$. Then the initiating node updates its successor list: $successor_j(u_i) := v$. The value $successor_1(u_i)$ is always defined by the Re-Chord stabilization process.

An illustration can be found in Fig. 6. In Fig. 6a) *find successors messages* are sent out and the found successors reply with *successor rely messages*. In Fig. 6b) u_i has created the corresponding $successor_j(u_i)$ edges.

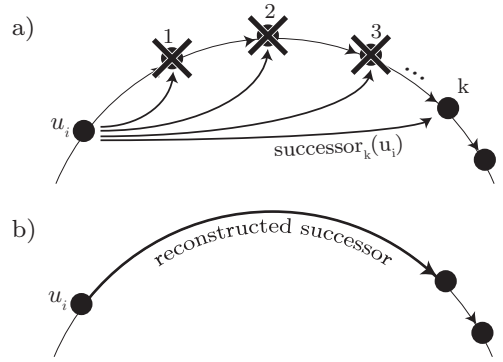


Fig. 7. Fast Repair: Up to $k-1$ Contiguous Nodes can Leave

C. Fast Repair Routine

A lost connection can be detected in two ways:

- Ping messages are sent periodically every T_{ping} simulation steps to the neighbors. If a node does not answer, the node is considered as offline.
- A message has been sent to a neighbor but no *acknowledge* message was received in a certain time interval. Then the node is considered as offline.

In both cases, the neighbor is removed from the edge-sets E_u, E_c and E_r. If the node is a *real successor* (real node) respectively a *real right neighbor* (virtual node), $r_r(u_i)$ is replaced by $successor_2(u_i)$ and $successor_2(u_i)$ is added to the unmarked edge set $E_u(u_i)$. After that, all successors are shifted one position left in the $successor(u_i)$ list, see for that illustration 7. A formal pseudo-code can be found below.

FASTREPAIR(u_i, v_{lost})
for all $v \in S(u_i)$ **do**
  $E_u(v) := E_u(v) \backslash \{v_{lost}\}$
  $E_c(v) := E_c(v) \backslash \{v_{lost}\}$
  $E_r(v) := E_r(v) \backslash \{v_{lost}\}$
end for
if $v_{lost} = r_r(u_i)$ **then**
  for $j := 1$ **to** $k-1$ **do**
    $successor_j(u_i) := successor_{j+1}(u_i)$
  end for
  $r_r(u_i) := successor_1(u_i)$
  $E_u(u_i) := E_u(u_i) \cup \{successor_1(u_i)\}$
end if

FASTREPAIR(u_i, v_{lost}) is called due to message failure or ping timeout with the node u_i and the lost neighbor v_{lost}.

D. Effects of Fast Fixing

Note that the approach presented above works in two ways. First, the closed gap by adding the edge to the E_u set speeds up the stabilization process. Second, because the right real neighbor $r_r(u_i)$ is updated, a failed message can be retransmitted and will use the new edge instead of the offline node. Furthermore, because we consider both, real and virtual nodes, in the fast repair routine not only the Chord ring is fixed but the fingers as well:

- Fast fixing applied on a *real node* means to fix the Chord ring because the *real successor* is fixed.

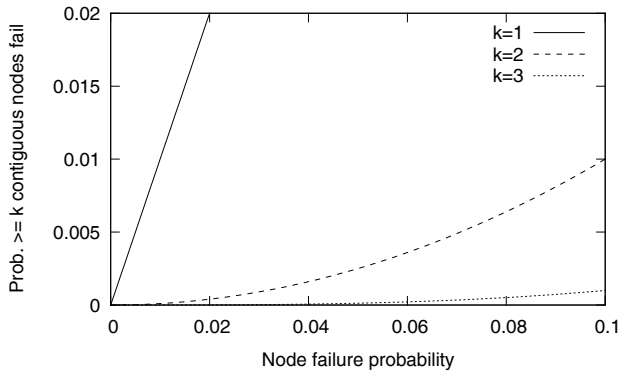


Fig. 8. Probability of Fast Repair Failure in Dependence of Node Failure

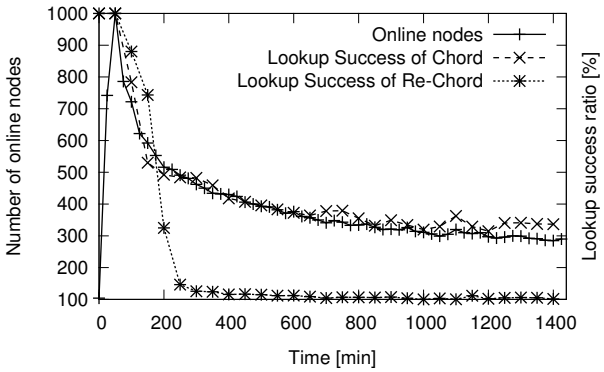


Fig. 9. Queries per Second in Re-Chord

- Fast fixing applied on a *virtual node* means to fix a finger of the Chord network as the *real right neighbor* is fixed.

If Ca-Re-Chord is in a stable state, the fast fix routine can fix up to $k-1$ contiguous failing nodes as it can be observed in Fig. 7. If Ca-Re-Chord is in a (heavily) non-stable state this fixing may fail due to incorrect data about successor nodes (outdated data structure $successor_j(u_i)$). It follows that Ca-Re-Chord is able to handle significantly more churn than Re-Chord but if churn is too high, the routing performance might get effected. Corresponding measurements can be found in Section V that confirm this observation. The number of stored successors k should be chosen according to the observed node failure rate, desired robustness against churn and acceptable communication overhead.

Theorem 6: Assume that each node has probability p for leaving the network in the actual round. Then the probability that the *fast repair routine* cannot fix the *real successor* (respectively *real right neighbor*) is p^k.

Proof: The *fast repair routine* cannot fix the successors if k contiguous nodes fail. The probability for failing is the product of the failing probability of each of the k nodes. Therefore, it follows p^k. ∎

An illustration for different values of k can be found in Fig. 8. In most cases, a low constant value between $k = 2$ and $k = 4$ will be a good choice and should be chosen in dependence on the desired *communication overhead/churn awareness* tradeoff. Note, that the failure rate of Fig. 8 does not mean that the network is disconnected but the *slower* self-stabilization has to be work instead of the fast repair routine.

V. Evaluation

The evaluation was performed with simulations that allow to investigate the characteristics of the protocols in large-scale networks. We implemented and evaluated Re-Chord and Ca-Re-Chord in the event-based simulator *PeerfactSim.KOM*, introduced in [6], [**?**], [11] and [17]. PeerfactSim.KOM is a Java based simulator that consists of many P2P protocols relating to applications, decentralized services and overlays, allowing multi-layer simulations up to several ten thousands of nodes. Peers were simulated to be located all over the world with Internet connection characteristics being distributed like determined by the OECD Bandwidth Report from 2007 [14]. The delay between the peers was determined by GNP [13]. Chord was implemented according to [19].

We simulated Chord, Re-Chord and Ca-Re-Chord under churn with exponentially distributed session times. First all nodes join and then they start leaving the network. The initial network size is 1000. Our churn model is based on Exponential Distribution. This distribution model is defined as follows. Let X be a positive real number belonging to $R_x = [0, +\infty)$. X has exponential distribution with mean distribution of μ if the probability density function is: $PDF(X) = (1/\mu)exp(-x/\mu)$. We used the exponential distribution as the churn model for session and inter session times. The churn factor is 0.5 and the mean session length 60 minutes. The number of successor edges in our Ca-Re-Chord simulations is 3.

A. Experimental Results

First we simulated Re-Chord under churn and identified in Fig. 9 Re-Chord's inability to cope with churn during the stabilization process. For that the nodes stored first 100 documents in the network from a Zipf-distributed file set (Zipf exp. 0.7) and retrieved them afterwards by initiating 2 lookups per peer per hour also following the same Zipf distribution. Churn has the effect that objects on left nodes are lost. While for Chord the lookup success ratio is tightly linked to the nodes in the network, for Re-Chord only a very small fraction of objects can be retrieved although the objects are still in the network. This is surprising, as the proofs show that Re-Chord can recover to a Chord network from any arbitrary state. However, during the stabilization procedure the lookup operation is broken in the case of more than 50% of nodes leaving the network.

Next, we simulated Ca-Re-Chord with low churn conditions to investigate the performance of the network when the churn rate is negligible. Fig. 10 shows the frequency for queries being started, successfully resolved and failed in Ca-Re-Chord network with 1000 nodes. The figure depicts that almost all of the queries are resolved by Ca-Re-Chord at moderate churn with churn factor 0.1. Successful queries could be a measurable factor in order to observe the potential of each overlay networks to serve the queries completely and successfully. Moreover, to make a precise comparison between the overlays during their runtimes, we use only the ratio of succeeded

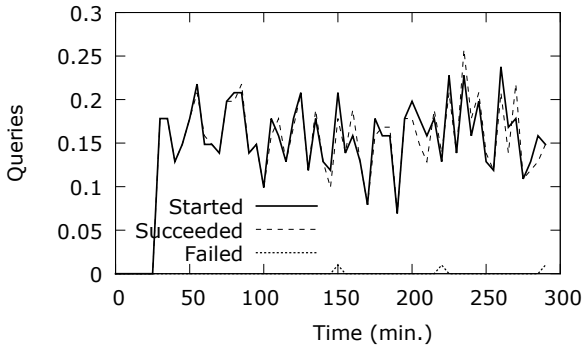


Fig. 10. Queries in Ca-Re-Chord. $ChurnFactor = 0.1$.

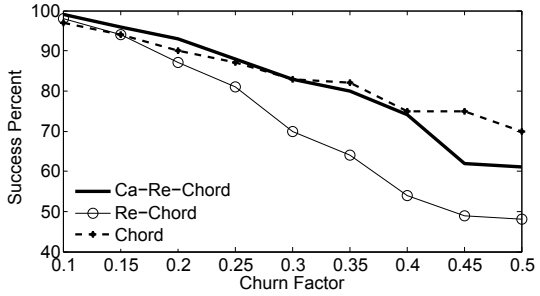


Fig. 11. Percent of Success in Chord, Re-Chord, and Ca-Re-Chord.

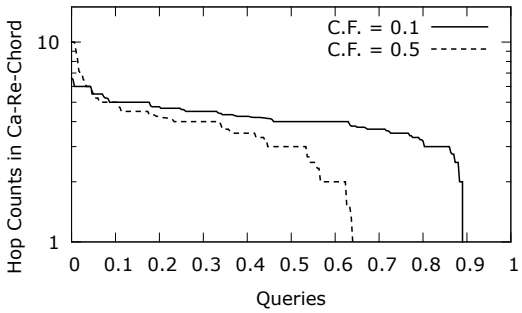


Fig. 12. Average number of hop counts with $ChurnFactor$ 0.1 and 0.5.

queries. We depict the results of the successful queries in Ca-Re-Chord under various churn in Fig. 11.

Fig. 11 proves that Ca-Re-Chord is able to handle the churn better than Re-Chord, while providing all functionalities of Chord and self-stabilization properties additionally. In this case, Ca-Re-Chord shows considerable efficiency comparing to Re-Chord. For high churn rates, when the network is under high rate of changes, the proposed algorithm provides results, which are more effective in comparison to Re-Chord. Even though Ca-Re-Chord resists to high churn rates remarkably, in a highly dynamic network Chord is still able to succeed in more lookups, most likely due to the less contacts to maintain. However, such networks are rare. The hop count is a considerable measurement for the performance of a network. The delay per hop can be improved easily [8]. In order to check the routing process in Ca-Re-Chord, we used the average number of hop count. As one can see in Fig. 12, the number of hops to get to the destination grows by increasing the churn factor, still in average the hop count is around 5, which is adequate for the network size as Chord performs alike.

VI. Conclusion

Although Chord has some attractive features considering simplicity and churn resistance, it cannot be recovered from any initial state in which the graph is weakly connected. To add this precious feature, a self-stabilizing distributed protocol *Re-Chord* is implemented using linearization techniques with local information only. Reactive Chord could guarantee to put any arbitrary, but weakly connected graph into the Chord state; however, in real scenarios, network dynamics hinder the stabilization process and make it unpredictable. Thus, in this paper we extend Re-Chord and propose *Ca-Re-Chord*, which is able to manage churn conditions as efficient as Chord and more efficient than Re-Chord. The simulation results demonstrate the effectiveness of the proposed Ca-Re-Chord protocol, while it is also self-stabilizing.

References

[1] L. O. Alima, S. Haridi, A. Ghodsi, S. El-Ansary, and P. Brand, "Position Paper: "Self-" properties in Distributed K-ary Structured Overlay Networks," in *Proc. of IEEE SELF-STAR '04*, 2004.

[2] A. Berns, S. Ghosh, and S. V. Pemmaraju, "Brief announcement: a framework for building self-stabilizing overlay networks," in *Proc. of ACM PODC '10*, 2010.

[3] J. Brzezinski, M. Szychowiak, and D. Wawrzyniak, "Self-Stabilization In Distributed Systems - A Short Survey," *Foundations of Computing and Decision Sciences*, vol. 25, no. 1, pp. 3–22, 2000.

[4] T. Clouser, M. Nesterenko, and C. Scheideler, "Tiara: A Self-stabilizing Deterministic Skip List," in *Proc. of IEEE SSS '08*, 2008.

[5] E. W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control," *Comm. of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[6] K. Graffi, "PeerfactSim.KOM: A P2P System Simulator - Experiences and Lessons Learned," in *Proc. of IEEE P2P '11*, 2011.

[7] K. Graffi, C. Groß, D. Stingl, D. Hartung, A. Kovacevic, and R. Steinmetz, "LifeSocial.KOM: A Secure and P2P-based Solution for Online Social Networks," in *Proc. of IEEE CCNC '11*, 2011.

[8] K. Graffi, K. Pussep, S. Kaune, A. Kovacevic, N. Liebau, and R. Steinmetz, "Overlay Bandwidth Management: Scheduling and Active Queue Management of Overlay Flows," in *Proc. of IEEE LCN '07*, 2007.

[9] T. Janson, P. Mahlmann, and C. Schindelhauer, "A Self-Stabilizing Locality-Aware Peer-to-Peer Network Combining Random Networks, Search Trees, and DHTs," in *Proc. of IEEE ICPADS '10*, 2010.

[10] S. Kniesburges, A. Koutsopoulos, and C. Scheideler, "Re-Chord: a Self-stabilizing Chord Overlay Network," in *Proc. of ACM SPAA '11*, 2011.

[11] A. Kovacevic, S. Kaune, H. Heckel, A. Mink, K. Graffi, O. Heckmann, and R. Steinmetz, "PeerfactSim.KOM - A Simulator for Large-Scale Peer-to-Peer Networks," Technische Universität Darmstadt, Germany, Tech. Rep. Tr-2006-06, 2006.

[12] N. Liebau, K. Pussep, K. Graffi, S. Kaune, E. Jahn, A. Beyer, and R. Steinmetz, "The Impact Of The P2P Paradigm," in *Proc. of AMCIS '07*, 2007.

[13] T. Ng, S. Eugene, and H. Zhang, "Global Network Positioning: A New Approach to Network Distance Prediction," *ACM SIGCOMM Comp. Comm. Review*, vol. 32, no. 1, pp. 61–61, 2002.

[14] OECD, "OECD Bandwidth Report," 2007. [Online]. Available: http://www.oecd.org/sti/ict/broadband

[15] M. Onus, A. Richa, and C. Scheideler, "Linearization: Locally self-stabilizing sorting in graphs," in *Proc. of ALENEX '07*, 2007.

[16] A. Richa, C. Scheideler, and P. Stevens, "Self-stabilizing de bruijn networks," in *Proc. of IEEE SSS '11*, 2011.

[17] D. Stingl, C. Groß, J. Rückert, L. Nobach, A. Kovacevic, and R. Steinmetz, "PeerfactSim.KOM: A Simulation Framework for Peer-to-Peer Systems," in *Proc of HPCS '11*, 2011.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of ACM SIGCOMM '01*, 2001.

[19] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions of Networking*, vol. 11, no. 1, pp. 17–32, 2003.