Theoretical Computer Science 512 (2013) 119–129

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Corona: A stabilizing deterministic message-passing skip list

Rizal Mohd Nor [a], Mikhail Nesterenko [b,*], Christian Scheideler [c]

[a] Department of Computer Science, International Islamic University Malaysia, Kuala Lumpur, Malaysia
[b] Department of Computer Science, Kent State University, Kent, OH, USA
[c] Department of Computer Science, University of Paderborn, Paderborn, Germany

ARTICLE INFO

Keywords:
Peer-to-peer networks
Overlay networks
Self-stabilization
Message-passing

ABSTRACT

We present Corona, a deterministic self-stabilizing algorithm for skip list construction in structured overlay networks. Corona operates in the low-atomicity message-passing asynchronous system model. Corona requires constant process memory space for its operation and, therefore, scales well. We prove the general necessary conditions limiting the initial states from which a self-stabilizing structured overlay network in a message-passing system can be constructed. The conditions require that initial state information has to form a weakly connected graph and it should only contain identifiers that are present in the system. We formally describe Corona and rigorously prove that it stabilizes from an arbitrary initial state subject to the necessary conditions. We extend Corona to construct a skip graph.

1. Introduction

In a peer-to-peer overlay network, each process can communicate with any other peer process over the underlying network as long as the process is aware of the peer's identifier. These identifier records form the network topology. Peer-to-peer networks are effective for distributed information storage, group communication and large scale computations. The amount of research literature on this subject is extensive [2–4,6,15,19,24,25,27].

The skip list [22] is a popular peer-to-peer topology as it allows efficient search and quick topology updates. Specifically, both identifier search as well as process deletion or addition in a skip list take $O(\log n)$ steps, where n is the number of nodes. A skip list may be either randomized or deterministic. While the randomized version may be simpler to implement, the deterministic one provides firm search and topology update bounds as well as greater assurance against failures, malicious behavior and unfavorable topology changes.

A skip list may not be sufficiently robust against node crashes. Indeed, a single node failure may disconnect the skip list. Neither is a skip list particularly suitable for concurrent searches. The standard measures of robustness and concurrency are expansion and congestion [4]. The expansion and congestion of the skip list are $O(1/n)$ and $\Omega(n)$ respectively. A skip list extension, the skip graph [3], significantly improves these metrics.

Peer-to-peer systems may include millions of nodes. At such scale, fault-tolerance and topology maintenance become a major concern. An optimistic failure recovery approach called self-stabilization [12] may be particularly suitable for peer-to-peer systems [1,21] as it is oblivious to the exact nature of the fault. Once the influence of the fault stops, regardless of the state in which this fault leaves the system, the self-stabilizing system is guaranteed to return to a correct state and remain in correct states thereafter.

Due to the large initial state space, self-stabilization programs require careful correctness proofs. If practical low atomicity communication models such as the message-passing system are considered, such proofs may become difficult both to

* Corresponding author. Tel.: +1 330 672 9109.
E-mail address: mikhail@cs.kent.edu (M. Nesterenko).

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2012.08.029

construct and to verify. Furthermore, a large initial state space may lead to excessive process memory demands during stabilization, especially during initial linearization: topological sorting of the processes [21].

Our contribution. We start the paper by determining the limits of existence of self-stabilizing solutions for peer-to-peer networks in message-passing systems. We prove that self-stabilization is possible only if the initial state information forms a weakly-connected graph and if all the process identifiers in the initial state are present in the system.

The main contribution of the paper is Corona: a self-stabilizing deterministic skip list construction algorithm in message-passing systems. To the best of our knowledge Corona is the first such algorithm. Subject to the proven necessary initial state restrictions, we show that Corona stabilizes and builds a *1-2* skip list.

Instead of struggling to counteract the large state space of message-passing systems, we are able to use the low-atomicity model to our advantage: the channels are employed as extra identifier storage space. This allows us to keep the Corona design relatively straightforward and to linearize processes using process memory that is independent of the system size. We extend Corona to build skip graphs and to accommodate topology updates.

Related literature. There is a large body of work on how to efficiently maintain peer-to-peer networks. Most of the results focus on preserving the overlay network in the legal set of states. Relatively few studies address the self-stabilization of such networks. Moreover, due to the topology being part of the system state, the majority of classic self-stabilizing techniques are not applicable to peer-to-peer networks.

Let us survey the publications in self-stabilization that specifically address peer-to-peer networks. A few papers address simple topologies. The Iterative Successor Pointer Rewiring Protocol [11] and the Ring Network [26] organize the nodes in a sorted ring. Onus et al. [20] linearize a network into a sorted linked list. However, they use a simplified synchronized communication model for their algorithm.

There are several studies of more sophisticated structures. Hérault et al. [16] describe a self-stabilizing spanning tree algorithm. Caron et al. [8] present a Snap-Stabilizing Prefix Tree for Peer-to-Peer systems while Bianchi et al. [7] show stabilizing peer-to-peer spatial filters. Dolev et al. [13] present a self-stabilizing hypertree. Clouser et al. [10] propose a deterministic self-stabilizing skip list for shared register communication model. Gall et al. [14] discuss models that capture the parallel time complexity of locally self-stabilizing networks that avoids bottlenecks and contention. Jacob et al. [23] generalize insights gained from graph linearization to two dimensions and present a self-stabilizing construction for Delaunay graphs. However, none of these structures approach the congestion and expansion of a skip graph. In another paper, Jacob et al. [17] present a self-stabilizing, randomized 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 a simple, synchronized message-passing model. In [5], 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, in the worst case, the construction of a clique. In that way, failures at the structure of the overlay network can easily be detected and repaired.

2. Model, notation and definitions

Peer-to-peer networks. A peer-to-peer overlay network program consists of a set N of n processes with unique identifiers. A process can communicate with any other peer process as long as it has a record of the peer's identifier. The communication is by passing messages through channels.

Peer-to-peer networks often require ordering the processes in a sequence according to their identifiers. Two processes a and b are *consequent*, denoted $\mathbf{cnsq}(a, b)$, if $(\forall c : c \in N : (c < a) \vee (b < c))$. That is, two consequent processes do not have an identifier between them. For the sake of completeness, we assume that $-\infty$ is consequent with the smallest id process in the system. Similarly, the largest id process is consequent with $+\infty$.

Graph terminology helps us in reasoning about peer-to-peer networks. A *link* is a pair of identifiers (a, b) defined as follows: either message $message(b)$ carrying identifier b is in the incoming channel of process a, or process a stores identifier b in its local memory. See Fig. 2 for an illustration of this. Note that a thus defined link is directed. In referring to such a directed link (a, b), we always state the predecessor process a first and the successor process b second. The *length* of a link (a, b) is the number of processes c such that $a < c < b$. Note that the length of (a, b) is zero if $\mathbf{cnsq}(a, b)$ is true. The length of $(-\infty, a)$ is zero if a is the smallest id in the system, it is n otherwise. Similarly, the length of $(b, +\infty)$ is zero if b is maximum and n otherwise. The *process connectivity* graph CP is the graph formed by the links of the identifiers stored by the processes. A *channel connectivity* multigraph CC includes both locally stored and message-based links. Self-loop links are not considered. By this definition, CP is a subgraph of CC. Note that besides the processes, CC and CP may contain two nodes $+\infty$ and $-\infty$ and the corresponding links to them. Graph CP captures current network connectivity information that all the processes keep. CC reflects the connectivity data that is stored implicitly in the messages in communication channels. Again, refer to Fig. 2 for an example of both graph types.

Computation model. Each process contains a set of variables and actions. A *channel* C is a special kind of variable whose values are sets of messages. We assume that the only information a message carries is process identifiers. We further assume that a message carries exactly one identifier. The identifiers are defined. That is, a message cannot carry $+\infty$ or $-\infty$. Channel message capacity is unbounded. Messages cannot be lost. The order of message receipts does not have to match the order of transmission. That is, the channels are not FIFO. Due to this, we treat all messages sent to a particular process as belonging to a single incoming channel.

An action has the form ⟨*guard*⟩ ⟶ ⟨*command*⟩. *guard* is either a predicate over the contents of the incoming channel or **true**. In the latter case the predicate and corresponding action are *timeout*. *command* is a sequence of statements assigning new values to the variables of the process or sending messages to other processes.

Program state is an assignment of a value to every variable of each process and messages to each channel. A program state may be arbitrary, the messages and process variables may contain identifiers that are not present in the network. An identifier is *existing* if it is present in the network. An action is *enabled* in some state if its guard is **true** in this state. It is *disabled* in this state otherwise. A timeout action is always enabled. We consider programs with timeout actions, hence, in every state there is at least one enabled action.

A *computation* is an infinite fair sequence of states such that for each state s_i, the next state s_{i+1} is obtained by executing the command of an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. We assume two kinds of fairness of computation: weak fairness of action execution and fair message receipt. *Weak fairness* of action execution means that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. *Fair message receipt* means that if the computation contains a state where there is a message in a channel, the computation also contains a later state where this message is not present in the channel.

We focus on programs that do not manipulate the internals of process identifiers. Specifically, a program is *compare–store–send* if the only operations that it does with process identifiers is comparing them, storing them in local process memory and sending them in a message. That is, operations on identifiers such as addition, radix computation, hashing, etc. are not used. In a compare–store–send program, if a process does not store an identifier in its local memory, the process may learn this identifier only by receiving it in a message. A compare–store–send program cannot introduce new identifiers to the network, it can only operate on the ids that are already there. If a computation of a compare–store–send program starts from a state where every identifier is existing, each state of this computation contains only existing identifiers.

A state *conforms* to a predicate if this predicate is **true** in this state; otherwise the state *violates* the predicate. By this definition, every state conforms to predicate **true** and none conforms to **false**. Let A and B be predicates over program states. Predicate A is closed with respect to the program actions if every state of the computation that starts in a state conforming to A also conforms to A. Predicate A converges to B if both A and B are closed and any computation starting from a state conforming to A contains a state conforming to B.

Problems. The *overlay network problem* maps each set of identifiers to a set of acceptable process connectivity graphs. For example, for every set of processes, the *linearization problem* specifies exactly one graph where each process is linked with its consequent processes.

Linearized overlay networks simplify process search. When discussing a linearized network, processes with identifiers greater than p are to the *right* of p, while processes with identifiers smaller than p are to the *left* of p. That is, we consider processes arranged in increasing order of identifiers from left to right. See Fig. 2 for an illustration.

The process search time in a simple linearized network is proportional to its size. This may not be acceptable in large-scale networks. Shortcut links are added to accelerate navigation. In a deterministic *skip list*, these links are created recursively by levels. The zero (bottom) level is the linearized list of processes. In a *k-l* skip list, a node a has a link to node b at level i if a and b are between k and l hops away at level $i-1$. For example, in a *1-2* skip list, a and b are linked at level i if they are no more than three and no less than two hops away at level $i-1$. Refer to Fig. 4 for an example of a *1-2* skip list.

In the *k-l skip list construction* problem, a set of processes is mapped to the set of possible skip lists. Note that in a linearization problem the set of identifiers uniquely determines the connectivity graph. In case of *k-l* skip list construction, depending on which processes participate at each level, the same list of identifiers may form several possible skip lists. Hence, the skip list construction problem specifies multiple acceptable *CP* graphs for a single set of processes.

We define the two problem properties below to aid us in formally stating the necessary conditions for the existence of a solution. An overlay network problem is *single component* if it maps every set of processes to a weakly connected process connectivity graph. Intuitively, a single component network overlay problem prohibits a program from separating the network into multiple components. The linearization and skip list construction problem are single component.

An overlay network problem $\mathcal{PG}$ is *disconnecting* if there is at least one set of processes S such that for every channel connectivity graph CP to which $\mathcal{PG}$ maps S, there is a cut set CS such that $|CS| < n-1$ which disconnects S. Note that such a cut set exists for any graph except for a completely connected one. Essentially, a disconnecting network overlay problem requires that in at least one case the desired channel connectivity graph is not completely connected. In other words, the program has to disconnect at least two processes. Naturally, both the linearization and skip list construction problem are disconnecting.

Problem solutions. A program $\mathcal{PG}$ *satisfies* or *solves* a problem $\mathcal{PR}$ from a predicate P if, for every set S, every computation of $\mathcal{PG}$ that starts in a state conforming to P contains a suffix with the following property. The channel connectivity graph CP is the same in every state of this suffix and this CP is one of the graphs to which $\mathcal{PR}$ maps S. That is, starting from the initial state in P, the solution has to implement at least one of the required CPs.

Program stabilization is *graph-identical* if every computation of a stabilizing program contains a suffix where CC contains the same links as CP. Such a program generates CC links that are already present in CP. If a process of such program receives a message, this message carries an identifier that the recipient process already stores and the process ignores the message.

A program is *unconditionally stabilizing* (or just *stabilizing*) if it solves the problem from $P \equiv \textbf{true}$. That is, every computation of a stabilizing program, regardless of the initial state, contains a correct suffix. Unconditional stabilization may be too strong for a program to possess. A program is *conditionally stabilizing* if $P \not\equiv \textbf{true}$. That is, such a program stabilizes from a limited set P of states.

We define two special cases of conditionally stabilizing programs. A program is *weakly channel-connectivity stabilizing* if it stabilizes only from the initial states where the channel-connectivity graph is weakly connected. A program is *existing identifier stabilizing* if it stabilizes only from states where every identifier is existing.

3. Necessary conditions

The necessary conditions stated in this section show that common overlay network topology specifications prohibit the existence of unconditionally stabilizing solutions. The necessary conditions are that initially the channel connectivity graphs need to be connected and non-existing identifiers are not present.

The proofs for these conditions rely on the lemma below. Intuitively, the lemma states that for the processes to form a connected topology they have to be at least weakly connected initially.

Lemma 1. *If a computation of a compare–store–send program starts in a state where the channel connectivity graph CC is disconnected, the graph is disconnected in every state of this computation.*

Proof. Let us consider, without loss of generality, a program state where the connectivity graph consists of two components C_1 and C_2. Assume the opposite: the computation starting from this state contains states where the two components of CC are connected. Let us consider the first such state s_1. In this state there must be two processes $a \in C_1$ and $b \in C_2$ that are neighbors. Assume the link is from a to b. That is, $(a, b) \in CC$.

Since s_i is the first connected state, this link does not belong to CC in the preceding state s_{i-1}. Since the program is compare–store–send, the new link cannot appear in the process memory, it must be due to a message sent to a by another process c in state s_{i-1}. A message to a carrying b can only be sent by a process c that has links to both a and b in s_{i-1}.

Since $(c, a) \in CC$, c belongs to the same component C_1 as a in s_{i-1}, and since $(c, b) \in CC$, c belongs to the same component C_2 as b in s_{i-1}. This means that C_1 and C_2 are weakly connected in a state s_{i-1} that precedes s_i. However, we assumed that s_i is the first state where the two components are connected. This contradiction proves the lemma. □

Theorem 1. *If a compare–store–send self-stabilizing program is a solution to a single-component overlay network problem, this program must be weakly channel-connectivity stabilizing.*

Proof. Assume the opposite. That is, there is a self-stabilizing program $\mathcal{PG}$ that solves a single-component overlay network problem $\mathcal{PR}$ and it is not necessarily weakly channel-connectivity stabilizing.

Since $\mathcal{PG}$ is a solution to $\mathcal{PR}$, for each set S, every computation of $\mathcal{PG}$ contains a suffix with the prescribed CP. Since $\mathcal{PG}$ is not necessarily weakly channel-connectivity stabilizing, this holds true for computations starting from a state where CC is disconnected. Program $\mathcal{PG}$ is a compare–store–send program. According to Lemma 1, if its computation starts from a state where CC is disconnected, it is disconnected in every state of this computation. Since CP is a subgraph of CC, it has to be disconnected in every state of this computation as well. However, $\mathcal{PR}$ is single-component. Since $\mathcal{PR}$ is single-component, it maps every set of processes S to a weakly connected process CP. This means that, contrary to our initial assumption, $\mathcal{PR}$ is not a solution to $\mathcal{PG}$. Hence the theorem. □

Theorem 2. *If a graph-identical compare–store–send program is a stabilizing solution to a single-component disconnecting overlay network problem, this program must be existing identifier stabilizing.*

Proof. Assume the opposite. Let $\mathcal{PG}$ be a compare–store–send program that is a graph-identical self-stabilizing solution to a single-component disconnecting overlay network problem $\mathcal{PR}$. Since $\mathcal{PR}$ is disconnecting, there is a set of processes S such that for every connectivity graph, there is a cut set that disconnects this graph.

Consider a computation σ of $\mathcal{PG}$ with set S. Let CP be the process connectivity graph to which this computation converges. Let CS be the cut set that separates S into two subsets S_1 and S_2. Since $\mathcal{PG}$ is graph-identical, σ contains a suffix where, in every state, CC has the same links as CP. Let s_1 be the first state of this suffix.

We examine a set of processes $S_1 \cup S_2$ and construct a new state of the program for this state as follows. The state of every process in $S_1 \cup S_2$ and its incoming channel is the same as in the initial state of σ. In addition, the incoming channels of each process a belonging to $S_1 \cup S_2$ in this state contain the messages that are sent to a by processes of CS in σ before state s_1. From this new state, we execute the actions of $\mathcal{PG}$ for processes $S_1 \cup S_2$ in the same sequence as in σ. The presence of messages from processes in CP allows us to do that. After this procedure we arrive at a state s_2. We then execute the actions of $\mathcal{PG}$ in arbitrary fair manner. Thus the constructed sequence is a computation of $\mathcal{PG}$.

Note that each process of $S_1 \cup S_2$ has the same state in s_1 and s_2. Since CS was a cut set of CP in s_1, there are no links between processes of S_1 and S_2 in either s_1 or s_2. This means that CP is disconnected in s_2. Graph CC has the same links as CP in s_1. This means that CC is disconnected in s_2 as well. According to Lemma 1, both CC and CP are disconnected in every state of this computation past s_2.

However, $\mathcal{PG}$ is supposed to be a solution to $\mathcal{PR}$. Problem $\mathcal{PR}$ is single-component. This means our constructed computation has to contain a suffix where CP is weakly connected in every state. This contradiction proves the theorem. □

```
process p
variables
    r,   // right identifier, greater than p
    l    // left identifier, less than p
actions
    message(id) ∈ p.C ⟶
        receive message(id)
        if id > p then
            if id < r then
                if r < +∞ then
                    send message(r) to id
                r := id
            else
                send message(id) to r
        if id < p then
            if id > l then
                if l > −∞ then
                    send message(l) to id
                l := id
            else
                send message(id) to l
    true ⟶
        if r < +∞ then send message(p) to r
        if l > −∞ then send message(p) to l
```

Fig. 1. Linearization component of Corona (l-Corona).

4. Linearization

Problem statement. In the linearization problem, each set of processes is mapped to the following process connectivity graph *CP*. Each process *p* in *CP* contains exactly two outgoing links: $p.r$ and $p.l$. The links conform to the following predicate *LP*:

$$(\forall a, b \in N : a < b : \mathbf{cnsq}(a, b) \Leftrightarrow ((a.r = b) \wedge (b.l = a)))$$

The predicate states that two processes are neighbors if and only if they are consequent. The linearization problem requires processes to keep links to consequent processes only. Hence the problem is disconnecting and the necessary conditions for the existence of the solution to disconnecting problems stated in Theorem 2 apply to linearization.

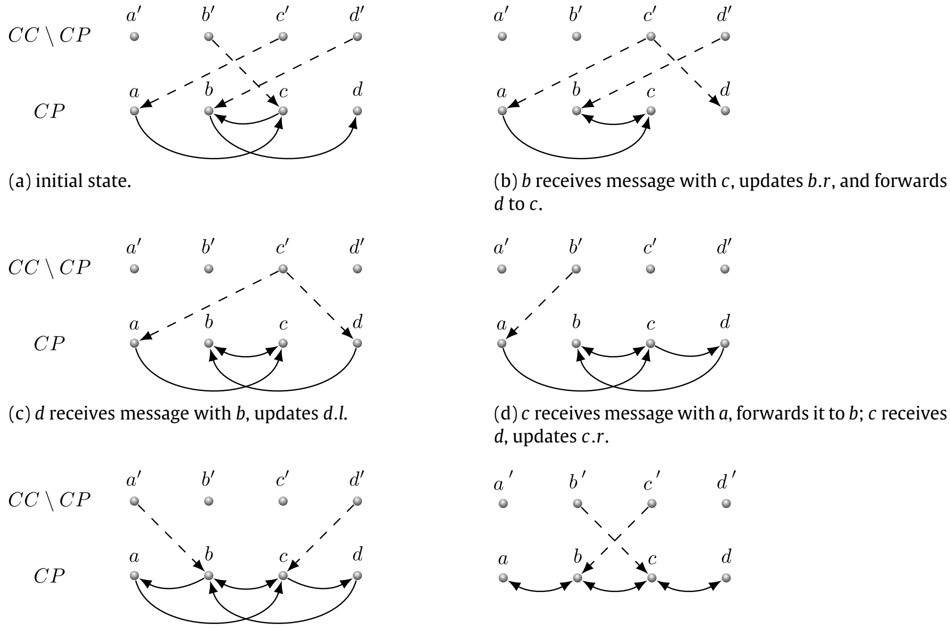
l-Corona description. Each process *p* maintains two variables *r* and *l* as required by the problem specification. The range of each variable are the process identifiers respectively to the left and to the right of *p*. That is, *r* can only store identifiers that are greater than *p*, while *l* – less than *p*. The value of each variable may be undefined. In this case it is equal to respectively $-\infty$ and $+\infty$. If non-existent identifiers are not present in the initial state of the program computation, the *l* variable of the smallest id process and the *r* variable of the largest id process are always set to $-\infty$ and $+\infty$ respectively.

The code of l-Corona is shown in Fig. 1. Each process *p* of l-Corona contains two actions: a receive action and a timeout action. The receive action is enabled when there is a message in the incoming channel $p.C$. The operation of the action depends on the *id* carried by the message. If *id* is greater than *p*, it is compared to *r*. If *id* is less than *r*, then *p* discovered a closer right neighbor. Process *p* then forwards the old right neighbor identifier to the new process and reassigns its variable *r*. However, if the received *id* is no less than *r*, then the current right neighbor of *p* is no further away than *id*. In this case *p* sends *id* to process *r*. If *r* is not initialized, it is assigned the received *id*. The identifier that is smaller than *p* is handled similarly. The timeout action sends the process identifier to its left and right neighbors. An example computation of l-Corona is shown in Fig. 2. By its operation of processes in l-Corona, they only compare, store and forward process identifiers. Hence, l-Corona is a compare–store–send program and the necessary conditions of Theorem 1 apply to it.

Correctness proof. We prove that l-Corona is weakly-channel connected and existing identifier stabilizing to the linearization problem. Therefore, throughout this subsection we assume that in every initial state, only existing identifiers are present and the channel connectivity graph is weakly connected.

Observe that due to the operation of the algorithm, in case $a < b$, link (a, b) can only be replaced by a link (a, c) such that $a < c < b$. Likewise, link (b, a) can only be replaced by (b, c) such that $a < c < b$. That is, a link in *CP* can only be shortened. An example of *CP* link shortening is shown in Fig. 2: the link (b, d) is shortened to (b, c) in transition from Fig. 2(a) to Fig. 2(b). Note that every process in *CP* contains exactly two outgoing links. One is pointing to the left, the other to the right.

Similarly, in case $a < b$, a link $(a, b) \in CC \setminus CP$ can be replaced only by a link (c, b) such that $a < c < b$. In the other direction, a link $(b, a) \in CC \setminus CP$ can be replaced only by a link (c, a) such that $a < c < b$. Again, the link in *CC* can only be shortened. For example, link $(c, a) \in CC \setminus CP$ in Fig. 2 is shortened to (b, a) in transition from Fig. 2(c) to Fig. 2(d). Note that unlike *CP*, a process may contain more than two outgoing links in $CC \setminus CP$. Furthermore, while some links are shortened, longer ones may be added by timeout actions.



(a) initial state.

(b) b receives message with c, updates $b.r$, and forwards d to c.

(c) d receives message with b, updates $d.l$.

(d) c receives message with a, forwards it to b; c receives d, updates $c.r$.

(e) b receives message with a, updates $b.l$; b times out and sends its id to a; c times out and sends its id to d.

(f) a receives message with b, updates $a.r$, forwards c to b; d receives message with c, updates $d.l$, forwards b to c; when these messages are received, the network is linearized.

Fig. 2. Example computation of l-Corona. To simplify the picture each process is represented by two nodes. The primed nodes are the process' incoming channel. Solid lines denote identifiers stored in l and r of each process. Dashed lines are identifiers in the incoming channel.

Lemma 2. *If a computation of l-Corona starts from a state where CC contains a path from process a to b, then in every state of this computation, there is a path from a to b as well.*

Proof. We show that the execution of every action of l-Corona either adds a link, retains all links, or replaces a link by a path. Therefore, none of the paths that contain these links before the action execution are disconnected by it.

Let us consider the receive action and focus on the identifier that the message carries. The self-loops are not considered in CC. Therefore, the case of $id = p$ is not applicable. We will only discuss the case of $id > p$, the case of $id < p$ is similar. If $r = +\infty$, the link is retained by p, and CC does not change.

Otherwise, this action of the program depends on the value of r. If $id > r$, then p forwards id to process r. That is, the link (p, id) is replaced by the path (p, r) and (r, id) in CC. Now, if id is between p and r, then p sends the value of r to id and updates the value of its right link to id. In other words, the link (p, id) is not changed in CC but link (p, r) is replaced by the path (p, id) and (id, r). Thus, the receive action of l-Corona does not disconnect paths in CC.

The case of the timeout action is straightforward as it only adds links to CC and thus cannot disconnect paths in CC. □

Lemma 3. *If a computation of l-Corona starts in a state where for some process a there are two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such that $a < c < b$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \le c$.*

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $b < c < a$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \ge c$.

Intuitively, Lemma 3 states that if there is a link in the incoming channel of a process that is shorter than what the process already stores, then the process' links will eventually be shortened.

Proof. We prove the lemma for the right neighbor of process a only. The proof for the left neighbor is similar. If link (a, c) belongs to $CC \setminus CP$, then there is a message-carrying c in the incoming channel of a. If link (a, b) belongs to CP, then the right neighbor of a is $b > c$.

Due to the fair message receipt assumption, the message carrying c will eventually be received by a. At the time of the message receipt, the right neighbor of a may be less than or equal to c, or greater than c. In the former case, the conclusion of the lemma is satisfied. In the latter case, once a receives c, it replaces its right neighbor with c. Hence, the lemma. □

Lemma 4. *If a computation of l-Corona starts in a state where for some process a there is an edge $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such that $a < b < c$, then the computation contains a state where there is a link $(d, c) \in CP$, where $d \le b$.*

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $c < b < a$, then this computation contains a state where there is a link $(d, c) \in CP$, where $d \ge b$.

Intuitively, the above lemma states that if there is a longer link in the channel, it will be shortened by forwarding the id creating this link to the id's closer successor.

Proof. We prove the lemma for the right neighbor of a. The condition of the lemma states that the right neighbor of a is b and there is a message-carrying $c > b$ in the incoming channel of a. Due to the fair message receipt assumption, the message-carrying c will eventually be received by a. Due to the operation of the algorithm, at the time of the message receipt, the right neighbor d of a is no greater than b. Since c is greater than b, a forwards c to d. The lemma follows. □

Lemma 5. *If a computation of l-Corona starts in a state where for some processes a, b, and c such that $a < c < b$ (or $a > c > b$), there are edges $(a, b) \in CP$ and $(c, a) \in CC$, then the computation contains a state where either some edge in CP is shorter than in the initial state or $(a, c) \in CP$.*

Proof. The timeout action in process c is always enabled. When executed, it adds *message*(c) to the incoming channel of process a. Then, the lemma follows from Lemma 3. □

Lemma 6. *If a computation starts in a state where there is a link $(a, b) \in CP$, then the computation contains a state where some link in CP is shorter than in the initial state or there is a link $(b, a) \in CP$.*

Proof. Assume without loss of generality that $a < b$. Once a executes its always enabled timeout action, link (b, a) is added to CC. We need to prove that either some link in CP is shortened or this link is added to CP.

Let us consider a link $(b, c) \in CP$ such that $c < b$. There can be three cases with respect to the relationship between a and c. In case $c < a$, the lemma follows from Lemma 3. In case $c = a$, the claim of the lemma is already satisfied. The case of $c > a$ is the most involved.

According to Lemma 4, if $c > a$, the computation contains a state where a shorter link to a belongs to CC. That is, there is a process d such that $a < d \leq c$ and $(a, d) \in CC$. Let us consider link $(e, d) \in CP$ such that $e < d$.

If $e < a$, then, according to Lemma 3, some link in CP shortens. If $e = a$, then some link in CP shortens according to Lemma 5. In both cases the claim of this lemma is satisfied.

Let us now consider the case where $e > a$. According to Lemma 4, the link to process a in CC shortens. The same argument applies to the new shorter link to a in CC. That is, either some link in CP shortens or a link to a shortens. Since the length of the link to a is finite, some link in CP eventually shortens. Hence the lemma. □

Lemma 7. *If the computation is such that if $(a, b) \in CP$ then $(b, a) \in CP$ in every state of the computation, then this computation contains a suffix where $((a, b) \in CP) \Rightarrow ((a, b) \in CC)$.*

Lemma 7 states that if CP does not change in a computation then eventually, the links in CP contain all the links of CC. The proof follows from the operation of the algorithm.

Lemma 8. *Let CP be strongly connected in some state of the system. Let it also be that for every pair of processes a and b in this state, if $(a, b) \in CP$ then $(b, a) \in CP$. In this case, this state satisfies LP.*

Proof. Let us prove the if part of LP first. Assume that the state in the condition of the lemma violates LP. That is, there is a pair of consequent processes u and v that are not neighbors. By condition of the lemma, CP is strongly connected. This means that there is a path from u to v. Let us consider the shortest such path. Since u and v are not neighbors, the path has to include processes to the left or to the right of both u and v. Assume without loss of generality $u < v$ and the path includes processes to the right of u and v. Let us consider the rightmost process in this path w. Let x and y be the processes that respectively precede and follow w in this path. Since w is the rightmost, both x and w are to the left of w.

Note that each process in CP can have at most one outgoing left and one outgoing right neighbor. By the condition of the lemma the outgoing neighbor of a process is also its incoming neighbor. Since x precedes w in the path from u to v and y follows w, x is the incoming and y is the outgoing neighbors of w. Yet, x and y are both to the left of w. This means that $x = y$. However, this also means that w can be eliminated from the path from u to v and can be shortened this way. However, we considered the shortest path from u and v. It cannot be further shortened. We arrived at a contradiction which proves the if part of the lemma.

The only if part follows from the observation that each process can only have a single right and single left neighbor. That is, if a process is already a neighbor with the consequent process it cannot be a neighbor with any other process. □

Theorem 3. *Program l-Corona is a weakly channel-connectivity existing identifier stabilizing solution to the linearization problem.*

Proof. To prove the theorem we show that l-Corona stabilizes to LP. The closure of LP follows immediately from the operation of l-Corona. Indeed, LP states that the links in CP connect consequent processes. The only change that l-Corona can do to links in CP is shorten them. However, the length of the links to consequent processes is already zero and they cannot be further shortened.

Let us now address the convergence of LP. Consider a computation of l-Corona. According to Lemma 6, for each process a if there is a link $(a, b) \in CP$, then some link is shortened in CP or there is a state where (b, a) also belongs to CP. Since links can be shortened only a finite number of times in a computation, there is a suffix of this computation where in every state if (a, b) belongs to CP so does (b, a). Note that CP does not change in this suffix of the computation, hence, according to Lemma 7, there is also a suffix where links in CP and CC are identical.

process p
constants
    $p.(i-1).r$, $p.(i-1).l$ // identifiers of right and left neighbors at level $i-1$
variables
    $p.i.st$, // own status at level i, either **up** or **down**
        // constant and set to **up** for process with highest id
        // constant and set to **down** for process with lowest id
    $p.i.str$ // status of right neighbor
actions
    $message(status) \in p.C$ **from** $p.(i-1).r \longrightarrow$
        receive $message(status)$,
        $p.i.str := status$,
        if $(p.i.st =$ **up**$) \wedge (p.i.str =$ **up**$)$ **then**
            $p.i.st :=$ **down**

    $message(status) \in p.C$ **from** $p.(i-1).l \longrightarrow$
        receive $message(status)$,
        if $(status =$ **down**$) \wedge (p.i.st =$ **down**$) \wedge (p.i.str =$ **down**$)$ **then**
            $p.i.st :=$ **up**

    true $\longrightarrow$
        if $p.(i-1).r < +\infty$ **then send** $message(p.i.st)$ **to** $p.(i-1).r$,
        if $p.(i-1).l > -\infty$ **then send** $message(p.i.st)$ **to** $p.(i-1).l$

Fig. 3. Status decision component of skip list part of Corona (sd-Corona).

According to Lemma 2, *CC* is not disconnected during a computation of l-Corona. This means that in this suffix *CP* is also connected. According to Lemma 6 then, *CP* is strongly connected. Then, according to Lemma 8, this computation contains a state where *LP* is satisfied. Hence the theorem. □

5. Skip list stabilization

Problem statement. The problem maps each set of processes to a set of valid *1-2* skip lists. In each skip list the bottom level is linearized and for each level $i > 0$, the following predicate *SL* holds: any two processes a and b are neighbors at level i if the distance between a and b at level $i-1$ is no less than 2 and no more than 3 hops. The *1-2* skip list construction problem requires a limited number of processes to be connected. For example, in the system of more than 3 processes, the processes with the largest and the smallest identifiers are not connected to each other. That is, the problem is disconnecting and the necessary conditions of Theorem 1 apply to it.

s-Corona description. The code of s-Corona is shown in Fig. 3. Each level of s-Corona has two sublevels: *status decision* sublevel – sd-Corona – and *neighbor linking* sublevel – sn-Corona.

sd-Corona of level i uses neighborhood information of level $i-1$ to determine the *status* of a process at level i. Depending on whether the process participates at level i, the process status is either **up** or **down**. If a process is **down** at level i it is **down** at all levels above i.

On the basis of this information, sn-Corona links p with its left and right neighbor at level i. Specifically, each process in sn-Corona at level i maintains three-hop neighborhood information of level $i-1$. This maintenance is done by each process periodically sending its immediate neighborhood information to the neighbors and attaching the hop count. All processes record this information and propagate it up to three hops away. If a process learns that it has obsolete or incorrect neighborhood information, before updating the information, the process sends itself this obsolete link at level 0 for l-Corona to handle. This ensures the overall *CC* connectivity preservation.

If process p at level i is **up**, sn-Corona inspects this locally stored three-hop neighborhood information to determine the nearest **up** neighbor and connects it to p.

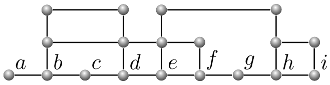
This stabilizing implementation of sn-Corona is relatively straightforward. We, therefore, do not present it in detail and focus on sd-Corona instead.

sd-Corona description. sd-Corona operates similarly at each level. At every level it maintains a set of variables that belong to only this level. At level i, process p of sd-Corona makes use of the identities $p.(i-1).l$ and $p.(i-1).r$ of its respective left and right neighbors at level $i-1$. sd-Corona at level i does not change these identities. Therefore, they are assumed constant for the operation of sd-Corona at this level.

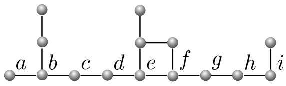
At level i, process p of sd-Corona maintains two status variables: $p.i.st$ and $p.i.str$. The values for both are **up** and **down**. Variable $p.i.st$ stores the status of p itself. Variable $p.i.str$ keeps the status of the right neighbor of p. The status of the rightmost and leftmost process at level i are fixed as **up** and **down** respectively and are considered constant.

The idea of sd-Corona is to ensure that no two consequent neighbors are **up** and no three of them are **down**. To break symmetry in deciding who of the neighbors should change status, the decision of the right neighbor is favored.

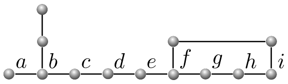
sd-Corona has three guards. The timeout guard sends the status of p to its neighbors. The two receive guards process messages from the left and right neighbors of p. If p receives a status value from its right neighbor, it updates $p.i.str$ and its own status. If both p and its right neighbor are **up** then p changes its status to **down**. If p receives a message from its left



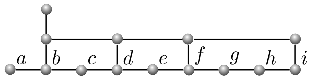
(a) initial state.



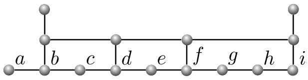
(b) at level 0, processes d and h receive messages that their right neighbors are **up**, they change their statuses to **down**.



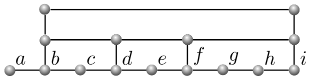
(c) at level 0, e receives message from f that its status is **up** and changes its own status to **down**; f and i are linked at level 1.



(d) at level 0, d receives messages that both c and e are **down** and changes its status to **up**, links with neighbors at level 1.



(e) at level 1, i receives message from f that its status is **down**, updates its own status to **up**.



(f) at level 2, i links with b.

Fig. 4. Example computation of s-Corona. For simplicity, neighbor links are always assumed bidirectional.

neighbor and discovers that its neighbors and itself are **down**, it changes its own status to **up**. The operation of s-Corona is illustrated in Fig. 4. Observe that, like the linearization component, s-Corona is a compare–store–send program and, as per Theorem 2, can only stabilize if the initial state only contains existing identifiers.

Correctness proof

Lemma 9. *If process a at level i of sd-Corona changes its status st only a finite number of times in the computation, then this computation contains a suffix where every message in the outgoing channel of a carries the same value as* $a.i.st$ *and* $b.i.str = a.i.st$ *for the left neighbor b of a.*

Proposition 1. *If, in some computation, none of the processes at some level i change their status, then this computation also contains a suffix where for each process a,* $a.i.r$ *and* $a.i.l$ *point to the nearest up process at this level and do not change.*

Lemma 10. *If in some computation none of the processes at some level* $i - 1$ *change their right and left neighbors, then this computation also contains a suffix where none of the processes at level i change their status.*

Proof. The proof is by induction on the number of processes on level i. The induction is carried out from the right end of the process list. To simplify the description we assume the processes are numbered 1 to n from right to left. Note that the status of the first (rightmost) process is constant. Assume that there is a suffix of the computations where $j - 1$ right processes do not change their status.

According to Lemma 9, this computation also contains a suffix where all messages from process $j - 1$ to process j, as well as $j.r$ have the same value as the status of process $j - 1$. In this case there is a suffix of the computation, where $j.i.r$ does not change. Then, in this suffix $j.i.st$ may change at most once. Specifically, if $j.i.st$ and $j.i.r$ are both **down**, then $j.i.st$ can be set to **up** if j receives a message with *status* = **down** from process $j + 1$. Thus, this computation contains a suffix where j does not change its status. The lemma follows by induction. □

Lemma 11. *In each computation of s-Corona, every process p changes its status and its left and right neighbors only finitely many times.*

Proof. The proof is by induction on the levels of s-Corona. At level zero, the lemma holds due to Theorem 3. Assume that there is a suffix of this computation where the status and neighbors of processes at level $i-1$ do not change. Then, according to Lemma 10, there is a suffix of this computation where the status of processes at level i does not change either. If that is the case, then, due to Proposition 1, there is also a suffix where the neighbors do not change. The lemma follows by induction. □

Theorem 4. *s-Corona is a weakly channel-connectivity existing identifiers stabilizing solution to the 1-2 skip list construction problem.*

Proof. To prove the theorem, we show that s-Corona converges to the *1-2* skip list predicate *SL*. According to Lemma 11, the processes in sd-Corona change their status only finitely many times.

Due to the algorithm design, this means that the sd-Corona converges to predicate where two consequent processes at level $i - 1$ cannot be **up** and three consequent ones cannot be **down**. That is, the process status at level i is appropriate for the *1-2* skip list. Due to Proposition 1 they are correctly linked. Hence the theorem. □

6. Extensions

Topology updates

A topology update is a node joining or leaving the set of processes N. We address topology updates when the system is in the correct state, i.e., we consider the simple case where a node joins or leaves a linearized set of processes. We do not address the stabilization of the topology update procedure.

Formally, we assume that in the initial state of the computation, the program satisfies the linearization predicate *LP*. That is, we consider re-constructing the skip-list above the linearized set of processes in case a single node is removed or added to this set.

When determining the status of level i only once the flags w.r.t. level $i-1$ have been set, a joining node will only start getting integrated in level i once it finds its correct place in level $i-1$. The following lemma can be proven by analyzing the several cases of adding or removing the node from level $i-1$.

Lemma 12. *A removal or addition of a node at level $i-1$ leads to at most one process status change in sd-Corona at level i.*

Proposition 2. *The operation of sn-Corona at level i in case of a single status change of a node in sd-Corona at level i is equivalent to a single state transition that reconnects **up** neighbors at level i.*

Recursively applying Lemma 12 and Proposition 2 to the levels of the skip list, we obtain the following theorem.

Theorem 5. *The number of topological changes Corona requires to reconstruct the skip graph after a single topology update is in $O(\log n)$.*

Skip graphs. Let us describe the extension of Corona to skip-graph. The skip list may not be robust or convenient for concurrent searches. Indeed, a failure of a single top-level node may disconnect the system. In a *k-l skip graph* [3], the processes at level $i-1$ that do not participate at level i form an alternative list at level i. If processes do not participate in this alternative one, they form the next one and so on. The process continues recursively both at the main as well as at the alternative list. That is, each list splits into several lists at each level. This way, most nodes have links at all levels of the skip graph. This property makes skip graphs more robust and better suited for concurrent searchers than skip lists.

Corona can be extended to construct a skip-graph. For that, Corona has to run two instances of sn-Corona at each level i. The main instance operates as before, while the alternative instance constructs an alternative list out of the nodes that do not participate in the main list. Note that in the *1-2* skip list, one alternative list can always be constructed. An instance of sd-Corona at level $i+1$ runs each of the lists. The process of splitting into main and alternative lists continues iteratively on each thus formed list. No changes are required in either l-Corona or sd-Corona.

***k-l* skip list.** Corona can be extended to accommodate an arbitrary *k-l* skip list in several ways. For example, each process in the extended version of Corona maintains the status of $k-1$ right neighbors and one left neighbor. If p detects that it is **up** and there is an **up** right neighbor less than l hops away, then p changes its status to **down**. If p is **down** and there are $k+1$ consequent **down** processes, it goes **up**.

7. Future work

In closing we would like to outline a couple of interesting research directions that extend our work to further its applicability and significance. We established that the presence of non-existent identifiers and lack of connectivity make self-stabilization impossible in asynchronous systems. This opens the question as to what is the minimal oracles, similar to Chandra and Toueg's failure detectors [9], that enable self-stabilization. Corona seems to be able to handle topological changes within rather limited locality. It would be interesting to see if Corona can be extended to handle churn [18]: stabilization during topology updates.

Acknowledgments

The third author was supported in part by DFG awards SCHE 1592/1-1 and SFB 901 (On-the-Fly Computing).

References

[1] Luc Onana Alima, Seif Haridi, Ali Ghodsi, Sameh El-Ansary, Per Brand, Position paper: Self-.properties in distributed k-ary structured overlay networks, in: Proceedings of SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems, in: Lecture Notes in Computer Science, vol. 3460, Springer, 2004.
[2] David Andersen, Hari Balakrishnan, Frans Kaashoek, Robert Morris, Resilient overlay networks, in: SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA, 2001, pp. 131–145.
[3] James Aspnes, Gauri Shah, Skip graphs, ACM Transactions on Algorithms 3 (4) (2007) 37:1–37:25.
[4] Baruch Awerbuch, Christian Scheideler, The hyperring: a low-congestion deterministic data structure for distributed environments, in: SODA'04: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004, pp. 318–327.

[5] A. Berns, S. Ghosh, S. V. Pemmaraju, Brief announcement: a framework for building self-stabilizing overlay networks, in: Proc. of the 29th ACM Symp. on Principles of Distributed Computing (PODC), 2010, pp. 398–399.
[6] Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, Mark Thober, Pagoda: a dynamic overlay network for routing, data management, and multicasting, in: SPAA'04: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, NY, USA, 2004, pp. 170–179.
[7] Silvia Bianchi, Ajoy Datta, Pascal Felber, Maria Gradinariu, Stabilizing peer-to-peer spatial filters, in: ICDCS'07: Proceedings of the 27th International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2007, p. 27.
[8] Eddy Caron, Frédéric Desprez, Franck Petit, Cédric Tedeschi, Snap-stabilizing prefix tree for peer-to-peer systems, Parallel Processing Letters 20 (1) (2010) 15–30.
[9] Tushar Deepak Chandra, Sam Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM 43 (2) (1996) 225–267.
[10] Thomas Clouser, Mikhail Nesterenko, Christian Scheideler, Tiara: A self-stabilizing deterministic skip list, in: Sandeep S. Kulkarni, André Schiper (Eds.), SSS, in: Lecture Notes in Computer Science, vol. 5340, Springer, 2008, pp. 124–140.
[11] Curt Cramer, Thomas Fuhrmann, Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe, 2005.
[12] Edsger W. Dijkstra, Self-stabilization in spite of distributed control, Communications of the ACM 17 (11) (1974) 643–644.
[13] Shlomi Dolev, Ronen I. Kat, Hypertree for self-stabilizing peer-to-peer systems, Distributed Computing 20 (5) (2008) 375–388.
[14] D. Gall, R. Jacob, A. Richa, C. Scheideler, S. Schmid, H. Täubig, Time complexity of distributed topological self-stabilization: The case of graph linearization, 2010, pp. 294–305.
[15] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, Alec Wolman, Skipnet: a scalable overlay network with practical locality properties, in: USITS'03: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems, Berkeley, CA, USA, 2003. USENIX Association, pp. 9–9.
[16] Thomas Hérault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, Joffroy Beauquier, Brief announcement: Self-stabilizing spanning tree algorithm for large scale systems, in: Ajoy Kumar Datta, Maria Gradinariu (Eds.), SSS, in: Lecture Notes in Computer Science, vol. 4280, Springer, 2006, pp. 574–575.
[17] R. Jacob, A. Richa, C. Scheideler, S. Schmid, H. Täubig, A distributed polylogarithmic time algorithm for self-stabilizing skip graphs, in: Proc. of the 28th ACM Symp. on Principles of Distributed Computing (PODC), 2009, pp. 131–140.
[18] F. Kuhn, S. Schmid, R. Wattenhofer, A self-repairing peer-to-peer system resilient to dynamic adversarial churn, in: Proc. of the 4th International Workshop on Peer-To-Peer Systems (IPTPS), 2005.
[19] Dahlia Malkhi, Moni Naor, David Ratajczak, Viceroy: a scalable and dynamic emulation of the butterfly, in: PODC'02: Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM, New York, NY, USA, 2002, pp. 183–192.
[20] Melih Onus, Andrea Richa, Christian Scheideler, Linearization: Locally self-stabilizing sorting in graphs, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2007.
[21] Melih Onus, Andréa W. Richa, Christian Scheideler, Linearization: Locally self-stabilizing sorting in graphs, in: ALENEX 2007: Proceedings of the Workshop on Algorithm Engineering and Experiments, SIAM, 2007.
[22] William Pugh, Skip lists: A probabilistic alternative to balanced trees, Communications of the ACM 33 (6) (1990) 668–676.
[23] C. Scheideler, R. Jacob, S. Ritscher, S. Schmid, A self-stabilizing local delaunay graph construction, in: 20th Intl. Symp. on Algorithms and Computation (ISAAC), 2009.
[24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Schenker, A scalable content-addressable network, in: SIGCOMM'01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, New York, NY, USA, 2001, pp. 161–172.
[25] Antony I. T. Rowstron, Peter Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Springer-Verlag, London, UK, 2001, pp. 329–350.
[26] Ayman Shaker, Douglas S. Reeves, Self-stabilizing structured ring topology P2P systems, in: Proc. 5th IEEE International Conference on Peer-to-Peer Computing, 2005, pp. 39–46.
[27] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, Chord: a scalable peer-to-peer lookup protocol for Internet applications, IEEE/ACM Transactions on Networking 11 (1) (2003) 17–32.