A Self-Stabilization Process for Small-World Networks

Sebastian Kniesburges
Department of Computer Science
University of Paderborn
Email: seppel@upb.de

Andreas Koutsopoulos
Department of Computer Science
University of Paderborn
Email: koutsopo@mail.upb.de

Christian Scheideler
Department of Computer Science
University of Paderborn
Email: scheideler@mail.upb.de

***Abstract*—Small-world networks have received significant attention because of their potential as models for the interaction networks of complex systems. Specifically, neither random networks nor regular lattices seem to be an adequate framework within which to study real-world complex systems such as chemical-reaction networks, neural networks, food webs, social networks, scientific-collaboration networks, and computer networks. Small-world networks provide some desired properties like an expected polylogarithmic distance between two processes in the network, which allows routing in polylogarithmic hops by simple greedy routing, and robustness against attacks or failures. By these properties, small-world networks are possible solutions for large overlay networks comparable to structured overlay networks like CAN, Pastry, Chord, which also provide polylogarithmic routing, but due to their uniform structure, structured overlay networks are more vulnerable to attacks or failures. In this paper we bring together a randomized process converging to a small-world network and a self-stabilization process so that a small-world network is formed out of any weakly connected initial state. To the best of our knowledge this is the first distributed self-stabilization process for building a small-world network.**

***Index Terms*—small-world network; self-stabilization; distributed algorithms**

I. Introduction

Complex networks have attracted a great deal of attention in the past years. It has turned out that a number of universal features are common in various real-world networks; e.g. many of these networks show a small-world property. Small-world properties are found e.g. in social/friendship graphs, road maps, food chains, electric power grids, metabolite processing networks and networks of brain neurons [1]. Small-world networks are named in analogy to the small-world phenomenon firstly proposed by Milgrim [17] in 1967, where six degrees of separation were observed in friendship graphs. Small-world networks have local properties like regular lattices, yet they also have small characteristic path lengths, i.e. the expected diameter as well as the average degree is small.

Also well-structured overlay networks have become a major topic in the field of distributed computing especially for Peer-to-Peer systems. Examples for such structured overlay networks are CAN [20], Pastry [21] and Chord [23]. These networks are used since by their construction they provide certain properties like efficient routing, load-balancing, small diameter and degree. They are designed for a large and highly dynamical setting with nodes that might join, leave or fail, but in general they do not provide a protocol to recover from any possible state. One solution for this problem is self-stabilization and some self-stabilizing protocols for overlay networks have already been presented in the recent years. Their properties make small-world networks an interesting alternative approach for overlay networks as they offer small routing distances compared to the afore mentioned structured overlay networks while having a low average degree. Additionally, small-world networks provide a certain robustness against failures or attacks [25]. We show in this paper that a self-stabilizing solution is not only possible for well-structured overlays but also for networks formed by a randomized process like a small-world network, in the sense that we still allow dynamics in the edges but show a convergence to and maintenance of desired network properties starting in any weakly connected state.

A. Small-World networks

It is known that by taking a connected graph or network with a high graph diameter and adding a very small number of edges randomly, the diameter tends to drop drastically. In such a way networks can be constructed which have the small world property [17]. In networks modeling friendships, the probability of befriending a particular person is assumed to be inversely proportional to the number of geographically closer people, which matches the experimentally observed small-world property [17],[16],[8]. The main mechanism to construct small-world networks is the Watts-Strogatz mechanism. In 1998, Watts and Strogatz [24] introduced a small-world network model that was capable of interpolating between a regular network and a random network using a single parameter. These graphs are constructed using a regular lattice, and on this lattice a process of constructing certain edges and probabilistic rewiring takes place, and a graph is formed that exhibits the small world properties. Watts and Strogatz argued that such a model captures two crucial parameters of social networks: there is a simple underlying structure that explains the presence of most edges, but a few edges are produced by a random process that does not respect this structure. They showed that a number of naturally arising networks exhibit this pair of properties. Kleinberg [14] showed based on the Watts-Strogatz model a mechanism not only constructing a small-

world network, but also allowing a distributed algorithm to find the shortest paths in this network. In particular Kleinberg showed that greedy routing performs polylogarithmically using a k-harmonic distribution [14] in a k-dimensional lattice.

The variation of the small-world network we use in this work is that presented in [4]. In this work a distributed and randomized process is described that leads to a k-harmonic distribution of the long-range links obtaining the same polylogarithmic performance like Kleinberg [14]. In this case, a graph G, given by a k-dimensional lattice, is enhanced with additional links chosen at random. More precisely, every node is given some long-range link pointing at another node in the graph. It turns out that for each long-range link added at a node u, the probability that the endpoint of this link is v is inversely proportional to the size of the ball of radius $distG(u, v)$ centered at u in G. So, in the k-dimensional lattice $\mathbb{Z}^k$, the probability that u has a long-range link pointing at v is essentially proportional to $1/d^k$ where d is the distance between u and v in the lattice. This setting of the long-range links enables greedy routing to perform in polylogarithmic expected number of steps (as a function of the distance in the lattice between the source and the target).

B. Self-Stabilization

As mentioned above, our algorithm is self-stabilizing. In the field of self-stabilization, researchers are interested in algorithms that are guaranteed to eventually converge to a desirable system state from any initial configuration. The idea of self-stabilization in distributed computing first appeared in a classical paper by E.W. Dijkstra in 1974 [7] in which he looked at the problem of self-stabilization in a token ring. Since Dijkstra's paper, self-stabilization has been studied in many contexts, including communication protocols, graph theory problems, termination detection, clock synchronization, and fault containment. For a survey see, e.g., [3], [9], [11].

Interestingly, though self-stabilizing distributed computing has received a lot of attention for many years, the problem of designing self-stabilizing networks has attracted much less attention. The universal techniques known for distributed computing in static networks (like logging) are not applicable here as they have not been designed to actively perform local topology changes (network changes are only considered as faults or dynamics not under the control of the algorithm). In [26] and [27] Dolev and Tzachar show self-stabilizing algorithms for forming subgraphs like clusters or expanders. In contrast to our model the challenge here is to choose the right subset of a (changing) set of edges and not to add new edges to reach a certain topology. Moreover, almost all algorithms considered until now work in a synchronous system. In order to recover scalable overlays from any initial graph, researchers have started with simple non-scalable line and ring networks. The Iterative Successor Pointer Rewiring Protocol [6] and the Ring Network [22] organize the nodes in a sorted ring. In [28] Dolev and Kat describe a strategy to build a hypertree with a polylogarithmic degree and search time. In [19], Onus et al. present a local-control strategy called linearization for converting an arbitrary connected graph into a sorted list. Clouser et al. [5] formulate a variant of the linearization technique for asynchronous systems in order to design a self-stabilizing skip list. In this work, our self-stabilization process has also as its basis a variance of the linearization technique. Gall et al. [10] discuss models that capture the parallel time complexity of locally self-stabilizing networks that avoids bottlenecks and contention. Jacob et al. [13] generalize insights gained from graph linearization to two dimensions and present a self-stabilizing construction for Delaunay graphs. In another paper, Jacob et al. [12] present a self-stabilizing variant of the skip graph and show that it can recover its network topology from any weakly connected state in $\mathcal{O}(\log^2 n)$ communication rounds with high probability. Recently, also a self-stabilizing variant of the popular Chord network was presented [15]. Moreover, in [18], the authors propose a self-stabilizing algorithm for skip list construction. The algorithm operates in a low atomicity message-passing asynchronous system model, which is the model we also adapt in our work. In [2] 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 may involves the construction of a clique. In that way, failures at the structure of the overlay network can easily be detected and repaired.

C. Our Contribution

In our work, we focus on the 1-dimensional variation of the graph presented in [4]. We propose a distributed self-stabilizing protocol that forms a ring which, in addition, contains long-range links as described above, giving us all the desired properties. In particular, the graph built by our protocol converges to a small-world network and, once established, the small-world properties are maintained. We further consider topology updates like joining and leaving nodes and show that the costs of a network recovery for such an update, counted in the number of messages sent, are polylogarithmic. To the best of our knowledge this is the first distributed self-stabilization process for building a small-world network.

II. Model

A. Network Model

We assume the following model for the overlay network which is similar to the model presented in [18] by Nor et al., but modified in some aspects to allow us to distinguish different edges and to form more complex networks than lists. The overlay network consists of N nodes or processes. An identifier $id \in [0, 1)$ is assigned to each node. Each node can communicate with any other node of the network as long as the other node's identifier is known. The nodes communicate in a standard message passing system by passing messages through channels. Each node has a local memory to store the identifiers of its neighbors. We say that if node u stores v's identifier the link or edge (u, v) exists. There are also temporary links that exist if u receives v's identifier in a message. A message contains a set of identifiers that is interpreted and used in

calculations by the receiving node and a type of the message to distinguish different reactions on the received messages. By this definition all links are directed and so is the formed graph. The length of a link (u, v) is the number of nodes w such that $u < w < v$ or $v < w < u$.

B. Computational Model

Each node contains a set of variables and performs different actions. We furthermore assume that each node has a channel C for incoming messages. We assume that the channel's capacity is unbounded and no messages are lost, but the order of the receipts does not have to match the order of transmission. For the channel we assume *fair message receipt* meaning that if there is a state in the computation where there is a message in the channel C there also is a later state where the message is not in the channel, but was received by the process. An action has the form $< guard > \rightarrow < command >$. *guard* is a predicate, that can be true or false. *command* is a sequence of statements, containing assignments of values, computations or sending messages to other nodes. The *program state* is defined by the assignment of values to every variable of each node and messages to every channel. An action is enabled in some state if its guard is true and disabled otherwise. A *computation* is a sequence of states such that for each state s_i the next state s_{i+1} is reached by executing an enabled action in s_i. By this definition, actions can not overlap and are executed atomically giving a sequential order of the executions of actions. For the execution of actions we assume *weak fairness* meaning that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. As in [18] we focus on programs that comply with the compare-store-send program model, in which programs do not manipulate the internals of the nodes' identifiers. Additionally we need further computations like additions and randomization to implement the move-and-forget mechanism presented in [4], but still these operations are not on the identifiers such that the presented programs are still compare-store-send programs.

III. Self-Stabilization Small-World Process

Next we define the actions each node executes and the used messages and internal variables. We distinguish between the following message types:

- *lin*: The standard message type to create links that are part of the so called linearization process.
- *inclrl*: This message type is used to mark incoming long range links that form the small-world network.
- *reslrl*: This message is sent to respond to an incoming long range link and to inform the origin of the long range link about possible network changes.
- *ring*: This type of message is used to establish a ring edge if a nodes misses its left neighbor.
- *resring*: This message is sent to respond to an incoming ring edge and to inform the source of the ring link about possible network changes.
- *probr*: This type of message is used to propagate the probing message
- *probl*: This type of message is used to propagate the probing message

Each node p has the following internal variables. For simplicity we assume the internal variables are always correct and can not be manipulated by an adversary, although the system can recover from corrupt internal variables by detecting them like wrong left or right neighbors $p.l$ and $p.r$ or resetting them over time $p.lrl$ and $p.age$ and $p.ring$:

- *p.id*: The identifier of the node p. We assume $p.id = p$.
- *p.l*: The identifier of p's left neighbor ($p.l < p$) or $-\infty$ if there is no left neighbor.
- *p.r*: The identifier of p's right neighbor ($p < p.r$) or ∞ if there is no right neighbor.
- *p.lrl* The identifier of the node that p's long range link points to.
- *p.ring*: The identifier of the node that p's ring edge points to. Note that this identifier is only set if $p.l = -\infty$ or $p.r = \infty$.
- *p.C*: The channel for incoming messages.
- *p.age*: The age of the long range given by the number of rounds since the last reset of $p.lrl$.

Each node has two different actions that we call the *receive* action and the *regular* action. The receive action is enabled if there is an incoming message on the channel $p.C$. Depending on the message type there will be different reactions. The regular action is enabled in every state, as its guard is *true*. Therefore there can be no state in the computation in which no action is enabled. In the regular action the node sends a message of type *lin* to its neighbors $p.l$ and $p.r$, a message of type *inclrl* to $p.lrl$, and if necessary a message of type *ring* in case that $p.l = -\infty$ or $p.r = \infty$. Additionally each node sends probing messages to ensure, that all nodes are in one connected component. The actions in pseudo code are presented in Algorithm 1.

A. Linearization

Our approach is based on a process called *linearization*, which has been used to form a sorted list in various papers [19][18]. The basic idea of this process is that each node p maintains two variables r and l where r stores an identifier greater than p and l an identifier smaller than p. If the value of $p.r$ is not defined, i.e. p does not know a greater node, it is set to ∞, and respectively $p.l$ might be set to $-\infty$. The values of $p.r$ (resp. $p.l$) are updated if p receives an identifier of a greater (resp. smaller) node smaller (resp. greater) than $p.r$ (resp. $p.l$). If the message is of type *lin* then the operation *linearize()* will be executed, in which the node tries to update its $p.l$ and $p.r$ entries by $m.id$ or forwards $m.id$ to $p.l$ or $p.r$. We extend this basic idea by using the long-range links as shortcuts when forwarding $m.id$ if $m.id > p.lrl > p.r$ (resp. $m.id < p.lrl < p.l$). The pseudo code is given in Algorithm 2. As p assumes $p.r$ and $p.l$ to be its consequent nodes, it has to inform them about its own id such that it can become $p.r$'s (resp. $p.l's$) direct neighbor, i.e. $p.r.l = p$ (resp. $p.l.r = p$). This is done in *lineraize(id)* presented in Algorithm 9.

B. Forming a ring

As the *move-and-forget* process requires not only a sorted list as a structure but a ring we need a further operation to form a ring. The basic idea of this procedure is that additional to $p.l$ and $p.r$ each node has the variable $p.ring$ to store a ring edge, that is an edge that connects the minimal node with the maximal node and vice versa. Thus these edges transform the sorted list to a sorted ring in $[0, 1)$. Obviously a ring edge only exists if either $p.l = -\infty$ or $p.r = \infty$. Note that only one of these cases can be true. Therefore in a stable sorted list only the maximal and minimal node create ring edges as all other nodes have a left and right neighbor. The ring edges are created in *linearize(id)* presented in Algorithm 9 in the regular action. If in the receive action a node receives a message containing a ring edge it responds such that the origin of the ring edge can update the endpoint. The response is presented in Algorithm 7 and the updating in Algorithm 8.

C. Probing

The probing procedure is introduced in order to ensure the connectivity in the network through non-long-range edges. Periodically, each time a specific time interval passes, each node sends a probing message to one of its neighbors. The purpose of this procedure is to test if the node is connected through paths without long-range links to the node, where its long-range link is currently positioned. We also use the probing procedure to ensure that a node and the node, where its ring edge is pointing to, are connected through non long-range links.

The procedure works as follows: A node p sends a probing message probr (resp. probl) along its edge to the right (resp. left) closest neighbor, if its long-range link $p.lrl$ is positioned on the right (resp. left) of p. W.l.o.g. let us from now on assume that the long-range link is positioned on the right (the procedure and proof for the left side is done similarly). The goal of the probing message is to reach its destination ($p.lrl$). So each time a probing message (probr) originating from a node p arrives at a node v, v forwards the message to the node $v.r$ or $v.lrl$ that is closest to $p.lrl$ and still smaller than $p.lrl$. In case of $p > p.lrl$ v receives a message of type probl and forward it to either $v.l$ or $v.lrl$. If the probing message fails to reach $p.lrl$ in that way (that is, if at any point there does not exist a right link, such that the distance to the destination is getting smaller), a link between the current node of the probing path and $p.lrl$ is created, and connectivity is assured. This procedure is also demonstrated in the algorithms. See Algorithm 10, Algorithm 6 and Algorithm 5.

D. Move and Forget

As we mentioned above, each node establishes links to other nodes. There exist two kinds of links, the links to the next known neighbors of the node and the ring edges that form the underlying ring network, as well as one long-range link for each node. The process that this long-range link follows is based on the Move-and-Forget Rewiring Process, described in [4]. The procedure works as follows: Assume a k-dimensional lattice $\mathbb{Z}^k$. In this lattice each node is initially occupied by exactly one token. These tokens move mutually independently according to random walks, starting from the node itself. That is, each token decides at each step its next position by altering its position in the lattice by ± 1 in each dimension with probability $\frac{1}{2}$. Nodes may forget their contacts through their long-range links. A long-range link of age $p.age = \alpha \geq 0$ (meaning that it is in existence for α steps) is forgotten with a probability $\phi(\alpha)$, where

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

Here, ϵ is a fixed (arbitrary small) parameter of the algorithm. Note that the probability $\phi(\alpha)$ is independent of the number of dimensions k When a long-range link is forgotten, the link stops existing and the token starts again its random walk from the original node.

We implement this process for the 1-dimensional case, that is the sorted ring. In the stable state, i.e. the ring is formed, the move process simplifies to choosing the right or left neighbor of the current endpoint of the long range link (each with probability 1/2). In the regular action in *sendid()* each node u informs the endpoint of the long-range link $u.lrl$ about the incoming link, which is presented in Algorithm 9. Then the informed node responds by sending the identifiers of its left and right neighbor to u. This is done in *repondlrl(id)* in the receive action presented in Algorithm 3. When node u receives the responses it performs the described move-forget process which is given in pseudo code in Algorithm 4. As already mentioned, the forget probability $\phi(\alpha)$ remains the same for the 1-dimensional case.

Algorithm 1 ACTIONS OF NODE P

```
message m ∈ p.C →
if m.type=lin then
    linearize(m.id)
else if m.type=inclrl then
    respondlrl(m.id)
else if m.type=reslrl then
    move-forget(m.id1,m.id2)
else if m.type=probr then
    probingr(m.id)
else if m.type=probl then
    probingl(m.id)
else if m.type=ring then
    respondring(m.id)
else if m.type=resring then
    updatering(m.id)
end if
true→
sendid()
probing()
```

Algorithm 2 LINEARIZE(ID)

if $id > p.id$ **then**
  if $id < p.r$ **then**
    if $p.r < \infty$ **then**
      send message (p.r,lin) to id
    end if
    p.r:=id
  else if $m.id > p.lrl > p.r$ **then**
    send message (id,lin) to p.lrl
  else
    send message (id,lin) to p.r
  end if
else if $id < p.id$ **then**
  if $id > p.l$ **then**
    if $p.l > -\infty$ **then**
      send message (p.l,lin) to id
    end if
    p.l:=id
  else if $m.id < p.lrl < p.l$ **then**
    send message (id,lin) to p.lrl
  else
    send message (id,lin) to p.l
  end if
end if

Algorithm 3 RESPONDLRL(ID)

if $p.l > -\infty \wedge p.r < \infty$ **then**
  send message(p.l,p.r,reslrl)
else if $p.l > -\infty \wedge p.r = \infty$ **then**
  send message(p.l,p.ring,reslrl)
else if $p.l = -\infty \wedge p.r < \infty$ **then**
  send message(p.ring,p.l,reslrl)
end if

Algorithm 4 MOVE-FORGET(ID1,ID2)

if $id1 > -\infty \wedge id2 < \infty$ **then**
  p.lrl:=id1 with probability $\frac{1}{2}$
  p.lrl:=id2 with probability $\frac{1}{2}$
else if $id1 > -\infty \wedge id2 = \infty$ **then**
  p.lrl:=id1
else if $id1 = -\infty \wedge id2 < \infty$ **then**
  p.lrl:=id2
end if
p.lrl=p.id with probability $\phi(p.age)$

Algorithm 5 PROBINGR(ID)

if $id \geq p.lrl \wedge p.lrl > p.r$ **then**
  send message (id,probr) to p.lrl
else if $id \geq p.r$ **then**
  send message (id,probr) to p.r
else if $p < id < p.r$ **then**
  linearize(id)
else
end if

Algorithm 6 PROBINGL(ID)

if $id \leq p.lrl \wedge p.lrl < p.l$ **then**
  send message (id,probl) to p.lrl
else if $id \leq p.l$ **then**
  send message (id,probl) to p.l
else if $p > id > p.l$ **then**
  linearize(id)
else
end if

Algorithm 7 RESPONDRING(ID)

if $id < p$ **then**
  if $p.l < id$ **then**
    send message (p.l,lin) to id
  else if $p.lrl < id$ **then**
    send message (p.lrl,lin) to id
  else if $p.lrl > p.r$ **then**
    send message (p.lrl,resring) to id
  else
    send message (p.r,resring) to id
  end if
else
  if $p.r > id$ **then**
    send message (p.l,lin) to id
  else if $p.lrl > id$ **then**
    send message (p.lrl,lin) to id
  else if $p.lrl < p.l$ **then**
    send message (p.lrl,resring) to id
  else
    send message (p.l,resring) to id
  end if
end if

Algorithm 8 UPDATERING(ID)

if $p.l = -\infty$ **then**
  if $id > p.ring$ **then** p.ring=id
  end if
else if $p.r = \infty$ **then**
  if $id < p.ring$ **then** p.ring=id
  end if
end if

Algorithm 9 SENDID()

if $p.l > -\infty$ **then**
  send message(p.id,lin) to p.l
else
  send message(p.id,ring) to p.ring
end if
if $p.r < \infty$ **then**
  send message(p.id,lin) to p.r
else
  send message(p.id,ring) to p.ring
end if
send message(p.id,inclrl) to p.lrl

Algorithm 10 PROBING()

```
if p.l = −∞ ∨ p.r = ∞ then
    if p.ring < p then
        if p.ring ≤ p.l then
            send message (p.ring,probl) to p.l
        else if p > p.ring > p.l then
            linearize(p.ring)
        else
        end if
    else
        if p.ring ≥ p.r then
            send message (p.ring,probr) to p.r
        else if p < p.ring < p.r then
            linearize(p.ring)
        else
        end if
    end if
end if
if p.lrl < p then
    if p.lrl ≤ p.l then
        send message (p.lrl,probl) to p.l
    else if p > p.lrl > p.l then
        linearize(p.lrl)
    else
    end if
else
    if p.lrl ≥ p.r then
        send message (p.lrl,probr) to p.r
    else if p < p.lrl < p.r then
        linearize(p.lrl)
    else
    end if
end if
```

IV. Analysis

A. Correctness

We show the correctness of our approach by proving that it stabilizes to a small-world network.

Theorem 4.1: The self-stabilizing small-world process provides a solution such that the graph eventually forms a one-dimensional small-world network.

We show this main theorem by dividing the self-stabilization process into different phases and determine the correctness of each phase. We show that the properties after one phase hold in each state afterwards once they are established. As shown above our approach is a *compare-store-send* program, such that theorems 1 and 2 of [18] also hold. We therefore assume that messages only contain existing identifiers and the graph is initially weakly connected.

We define the following graphs:

Definition 4.2: CC is the channel connectivity graph that contains all links of the network, i.e stored links like $(p, p.r)$, $(p, p.l)$, $(p, p.ring)$ or $(p, p.lrl)$ and implicit links given by the messages in the channel. We further define CP as the node connectivity graph, that is the graph formed by the stored identifiers. Note that $CP \subseteq CC$. According to the definitions of CC and CP we introduce further graphs:

- LCC is the list channel connectivity graph taking all links into account that take part in the linearization process. These links are formed by messages of type lin and the stored links to $p.r$ and $p.l$.
- LCP is the list node connectivity graph formed by the stored links to $p.r$ and $p.l$.
- RCC is the ring channel connectivity graph formed by LCC, the stored links $p.ring$ and all messages of type $ring$.
- RCP is the ring node connectivity graph formed by LCP and the stored links $p.ring$.

B. Phase 1: Connectivity

We show that after a first phase, if CC is weakly connected also LCC becomes weakly connected and thus it is possible to linearize LCC in the next phase.

Theorem 4.3: If the computation of the self-stabilizing small-world process starts in a state in which CC is weakly connected the computation contains a later state in which LCC is weakly connected and there are no links added to LCC by the *probing* operation, i.e. the probing succeeds in every following state.

We prove the theorem by showing some lemmas.

Lemma 4.4: If the computation of the self-stabilizing small-world process starts in a state in which a pair u and $u.lrl$ is not connected by a path of edges $(a, b) \in LCP$ where $u < a < b < u.lrl$ (resp. $u > a > b > u.lrl$), then the computation contains a later state such that u and $u.lrl$ are connected by a path of edges (a, b) where $u < a < b < u.lrl$ (resp. $u > a > b > u.lrl$).

Proof: We show Lemma 4.4 by induction. W.l.o.g. let $u < u.lrl$.

Inductive basis: We take u and $u.lrl$, such that there is no pair v and $v.lrl$ where $u < v < v.lrl < u.lrl$. Then the probing message follows a path in LCP. When the probing message reaches a node $u < w < u.lrl$ there can only be three cases:

1) $w.r < u.lrl$ then the probing message is forwarded to $w.r$ and $(w, w.r) \in LCP$.
2) $w.r = u.lrl$ then u and $u.lrl$ are already connected.
3) $w.r > u.lrl$ a link $(w, u.lrl)$ is added to LCP and u and $u.lrl$ become connected.

As the probing message can only be forwarded a finite number of times to the right neighbor, u and $u.lrl$ eventually are connected in LCP by a path of edges (a, b) where $u < a < b < u.lrl$.

Inductive Step: Let's consider the path a probing message can take from u to $u.lrl$. According to the probing operation,

probing messages are either forwarded from v to $v.r$ or $v.lrl$ as long as $v.r < u.lrl$ and $v.lrl < u.lrl$. The edges $(v, v.r)$ are always in LCC and by the inductive hypothesis there is path in LCP from v to $v.lrl$ by edges (a, b) where $v < a < b < v.lrl$. Analogous to the inductive basis eventually the probing message reaches a node w such that $w.r \geq u.lrl$ and u and $u.lrl$ become connected. ∎

Lemma 4.5: If the computation of the self-stabilizing small-world process starts in a state in which a pair u and $u.lrl$ are connected by a path of edges $(a, b) \in LCP$ where $u < a < b < u.lrl$ (resp. $u > a > b > u.lrl$), then there are no links added to LCP by unsuccessful probings.

Proof: According to the probing operation the probing message reaches $u.lrl$ and no link is created. ∎

We can state the same lemmas for ring edges, where the proof is analogous to the proofs given above as their connectivity is also checked by the probing.

Lemma 4.6: If the computation of the self-stabilizing small-world process starts in a state in which a pair u and $u.ring$ are not connected by a path of edges $(a, b) \in LCP$ where $u < a < b < u.ring$ (resp. $u > a > b > u.ring$), then the computation contains a later state such that u and $u.ring$ are connected by a path of edges (a, b) where $u < a < b < u.ring$ (resp. $u > a > b > u.ring$).

Lemma 4.7: If the computation of the self-stabilizing small-world process starts in a state in which a pair u and $u.ring$ are connected by a path of edges $(a, b) \in LCP$ where $u < a < b < u.ring$ (resp. $u > a > b > u.ring$), then there are no links added to LCP by unsuccessful probings.

Theorem 4.3 follows directly from Lemmas 4.4, 4.5, 4.6 and 4.7. As CC is weakly connected in the initial state it is weakly connected by edges in LCC, long-range links and ring links. As the latter two can eventually be substituted by paths from $LCP \subseteq LCC$ the network is weakly connected in LCC. Additionally the probing always succeeds once it has been successful.

C. Phase 2: Linearization

In the second phase we show that the network stabilizes to a state that contains the sorted list as a subgraph. We slightly modify the proof for the linearization problem given in [18].

Definition 4.8: In a *sorted list* each node p contains two outgoing links $p.l$ and $p.r$ such that the following condition holds: $\forall a, b \in V : a < b : b = \min c \in V : c > a \Leftrightarrow ((a.r = b) \wedge (b.l = a))$.

Theorem 4.9: The self-stabilizing small-world process provides a solution such that a subgraph formed by LCP is a solution to the sorted list problem.

To show the theorem we prove a set of lemmas from which the theorem follows. The main idea of the proofs is that the lengths of the links decrease until all consequent nodes are directly connected.

Lemma 4.10: If a computation of the self-stabilizing small-world process reaches a state where nodes a, b are connected by a path in LCC then in every state after that there is a path from a to b in LCC.

Proof: We show that in executed actions of the self-stabilizing small-world process links are kept, added or substituted by a path, such that the connectivity is maintained and no paths are disconnected.

1) receive action for message m: LCC is only altered by operations triggered by a message m if $m.type = lin$ or $m.type = ring$. The only operation executed for messages of type lin is *linearize(m.id)*. In *linearize(m.id)* the operations executed depend on the value of $m.id$. If $m.id > p.id$ and $m.id > p.lrl > p.r > p.id$ the link $(p, m.id)$ is replaced by a path $p, v_1, v_2, \cdots, p.lrl, m.id$ in LCC. The path $p, v_1, v_2, \cdots, p.lrl$ exists according to Theorem 4.3 and the link $(p.lrl, m.id)$ is created by a message $m'(m, id, lin)$ to $p.lrl$. If $m.id > p.r > p.id$ and $p.lrl < p.r$ then the link $(p.m.id)$ is replaced by a path $(p, p.r)(p.r, m.id)$ as p forwards $m.id$ to $p.r$. If $p.r = \infty$ the link $(p, m.id)$ is retained in LCC and even LCP. If $\infty > p.r > m.id > p.id$ then the link $(p, m.id)$ is retained and the link $(p, p.r)$ is substituted by a path $(p, m.id), (m.id, pr)$ as p sends a message $m'(p.r, lin)$ to $m.id$. The same arguments hold for the case $m.id < p.id$. If $m.type = ring$ and $m.id > p.id$ the node $a = m.id$ has no right neighbor ($a.r = \infty$) and if $p.lrl > m.id$ or $p.r > m.id$ a new link is added to LCC, but no path is disconnected.
2) Regular action: In the regular action links are only added to LCC, by either the *send* operation or in the *probing* operation, if p and $p.lrl$ are not connected.

∎

Lemma 4.11: If the computation of the self-stabilizing small-world process reaches a state where for some node a there are two links $(a, b) \in LCP$ and $(a, c) \in LCC - LCP$ such that $a < c < b$ (resp. $b < c < a$) then this computation contains a later state where a link $(a, d) \in LCP$ with $d \leq c$ (resp. $c \leq d$).

Proof: If $(a, c) \in LCC - LCP$ there is a message $m(c, lin) \in a.C$. Once this message is received by a, $a.r$ is either updated to c, such that $(a, d) \in LCP$ with $d = c$, or if a.r is not updated, $a.r$ must be set to $d < c$, such that $(a, d) \in LCP$ with $d < c$. ∎

This lemma claims that the stored links are shortened over time.

Lemma 4.12: If a computation of the self-stabilizing small-world process contains a state where for a node a there are links $(a, b) \in LCP$ and $(a, c) \in LCC - LCP$ then

the computation contains a later state where there is a link $(d, c) \in LCC$ with $a < d < c$.

Proof: If $(a, c) \in LCC - LCP$ there is a message $m(c, lin) \in a.C$. Once this message is received by a, we know by Lemma 4.11 that there is a link $(a, e) \in LCP$ with $a < e \leq b$ as the stored links in LCP are only shortened over time. Then c is forwarded by executing the *linearize(c)* operation to either $e = a.r$ or to $f = a.lrl$ by a message $m'(c, lin)$. For both $d = e$ and $d = f$ it holds that $a < d < c$. This lemma claims that links in the channel are also shortened over time, but with the difference that for these links the origin of the link and not its endpoint is updated. ■

Lemma 4.13: If a computation of the self-stabilizing small-world process contains a state where for some nodes a, b and c such that $a < c < b$ (resp. $b < c < a$) there are edges $(a, b) \in LCP$ and $(c, a) \in LCP$ then the computation contains a later state where either some edge in LCP is shorter or $(a, c) \in LCP$.

Proof: Remember that an regular action is always enabled. In this action c sends a message $m(c, lin)$ to a. Then by Lemma 4.11 this lemma is proven. ■

Lemma 4.14: If a computation of the self-stabilizing small-world process contains a state where there is an edge $(a, b) \in LCP$ then the computation contains a later state where some edge in LCP is shorter or $(b, a) \in LCP$.

Proof: Let $a < b$. In a regular action the edge (b, a) is added to LCC (if not already existing). If this edge is also added to LCP, that is $b.l = a$, then the lemma holds. Otherwise $a < c = b.l < b$ and according to lemma 4.12 the edge $(b, a) \in LCC$ is shortened to edge $(d, a) \in LCC$ where $a < d < b$. If $(d, e) \in LCP$ where $e < a$, then an edge in LCP shortens according to Lemma 4.11. If $e = a$, Lemma 4.13 holds and again an edge in LCP shortens. Otherwise according to Lemma 4.12 the edge to a shortens to (f, a) where $a < f < d$. The link in LCC can only shorten a finite number of times until a link in LCP has to be shortened. ■

Lemma 4.15: If a computation of the self-stabilizing small-world process is in a state such that $(a, b) \in LCP \Rightarrow (b, a) \in LCP$ in every state after that in the computation, then this computation contains a state after that LCP is strongly connected.

Proof: As Theorem 4.3 holds there is a state after which no edges are added to LCC by the *probing* operation. If $(a, b) \in LCP \Rightarrow (b, a) \in LCP$ holds then in a regular action no edges are added to LCC. Edges that are in $LCC - LCP$ are shortened over time according to Lemma 4.12 and LCC converges to LCP. As LCC is weakly connected again according to Theorem 4.3 and stays connected by Lemma 4.10 and $(a, b) \in LCP \Rightarrow (b, a) \in LCP$ holds, LCP becomes strongly connected. ■

Lemma 4.16: If a computation of the self-stabilizing small-world process is in a state such that LCP is strongly connected and for every pair of nodes $(a, b) \in LCP \Rightarrow (b, a) \in LCP$ then this state is a solution for the sorted list problem.

Proof: Let us assume that it is not a solution for the sorted-list problem. Then there is a pair a, b of consequent nodes that are not directly connected. As LCP is strongly connected there is a shortest path from a to b and vice versa. On this path there has to be a node c that has two outgoing edges to nodes $< d$ (or $> d$). This is a contradiction to the definition of LCP in which each node has only one outgoing edge to a node to the left and to the right. ■

We are now ready to show the Theorem 4.9. Obviously once a state is reached that is a solution for the sorted-list problem, no edges in LCP are changed as all consequent nodes are directly connected and thus no edge in LCP can be shortened. According to Lemma 4.14 there is a state such that in every state after $(a, b) \in LCP \Rightarrow (b.a) \in LCP$. Now, according to Lemma 4.15 the computation contains a later state such that LCP is strongly connected. Then by applying lemma 4.16 the computation contains a state with a solution for the sorted-list problem.

D. Phase 3: Forming a ring

In the third phase we show that the network stabilizes to states that contain the sorted ring as a subgraph.

Definition 4.17: In a *sorted ring* each node p contains two outgoing links $p.l$ and $p.r$ such that the following condition holds: $\forall a, b \in V : a < b : b = \min c \in V : c > a \Leftrightarrow ((a.r = b) \wedge (b.l = a))$. Additionally the minimal node min and the maximal node max contain ring edges pointing to each other, $min.ring = max$ and $max.ring = min$, closing the ring.

Theorem 4.18: The self-stabilizing small-world process provides a solution such that a subgraph formed by RCP is a solution to the sorted ring problem.

We prove the theorem by proving the following lemmas.

Lemma 4.19: If a computation of the self-stabilizing small-world process is in a state such that LCP is a solution for the sorted list problem then the computation contains a later state such that $LCC = LCP$.

Proof: It follows from Theorem 4.3 that edges are added To LCC only due to ring edges converted to edges in LCC. If LCP is a solution for the sorted list problem each node has a left and right neighbor, except the nodes min and max with the minimal or maximal identifier. Therefore only these two nodes store a ring edge. As there are no nodes $u < min$ or $u > max$, these ring edges never add links to LCC. According to the linearization analysis all edges in LCC are shortened over the time until a state is reached after which $LCC = LCP$. ■

Lemma 4.20: If a computation of the self-stabilizing small-world process is in a state such that LCP is a solution for the

sorted list problem and $LCC = LCP$ then the computation contains a later state such that $RCC = RCP$ and RCP is a solution to the sorted ring problem.

Proof: If LCP is a solution for the sorted list problem each node has a left and right neighbor, except the nodes min and max with the minimal or maximal identifier. Therefore only these two nodes have a ring edge in RCP. These ring edges are forwarded a finite number of times until eventually $min.ring = max$ and $max.ring = min$. Then as $LCC = LCP$, also $RCC = RCP$, and as LCP solves the sorted list problem, RCP solves the sorted ring problem. ■

E. Phase 4: A small-world network

In this last phase of the stabilization process we prove that in the end the self-stabilizing small-world process reaches a state in which it solves the small-world network problem. We follow the description of a small-world network given in [4].

Fact 4.21: A graph formed by the k-dimensional lattice $\mathbb{Z}^k$ in which each node has connections to its direct neighbors and each node has one additional long-range link, whose length follows the k-harmonic distribution,provides the small-world properties and is called a small-world network.

Theorem 4.22: The self-stabilizing small-world process provides a solution such that the graph formed by CP is a solution to the one-dimensional small-world network problem.

Proof: From Theorems 4.3, 4.9 and 4.18 it follows that the graph stabilizes to a graph that contains the $\mathbb{Z}^k$ for $k = 1$ as a subgraph in CP (the sorted ring). The edges not considered so far are in $CC - RCC$. These are the stored long-range links, links represented by incoming long-range link messages and responses to these messages and links represented by probing messages and their responses. The links represented by probing messages and their responses never stabilize as the connectivity is checked periodically. This also holds for the links represented by incoming long-range link messages and responses to these messages as long-range links change over time according to the move-and-forget process. Furthermore these links are only stored in messages and not in internal variables, so we can not show $CC = CP$, but only consider the graph given by CP. The only edges in $CC - RCC \cap CP$ are the stored long-range links. Once the computation reaches a state such that RCP forms the sorted ring, then there is a later state in the computation such that all long-range links have been forgotten at least once between these states. It can easily been shown by using properties shown in [4] that the maximal age of a long-range link is $\mathcal{O}(n)$ w.h.p.(with high probability is a probability $\geq 1 - \frac{1}{n^c}$). Then it can also been proven that after at most $\mathcal{O}(n)$ steps all long-range links have been forgotten at least once. After this state the move-and forget process for the long range links is performed on the $\mathbb{Z}^1$ and eventually CP forms a small-world network and maintains it forever according to [4]. ■

F. Efficiency

In this section we show that the probing procedure does not produce much overhead in form of messages as only polylogarithmic many hops and thus probing messages are necessary to ensure connectivity in the stable state.

Lemma 4.23: If the network is at a stable state, a probing message does not take more than $O(\ln^{2+\epsilon} d)$ hops to reach its destination, where d is the distance between the node and its long-range link.

Proof: At this point, we use the analysis done in [4] to compute the number of steps needed in order to complete the routing of a message in the small-world network. The authors claim that at each routing step, the distance to the destination is halved with a probability at least $\Omega\left(\frac{1}{ln^{1+\epsilon}d}\right)$. In other words, the probability that the next node of the routing path is a part of B, where B is the set of nodes, which have a distance to the destination node t which is smaller than half of the distance of the current node w (of the routing path) to the destination. $B = \{v \in \mathbb{Z}^k : dist(v,t) \geq dist(w,t)/2\}$ So, in our 1-dimensional case, B would consist of all the nodes being at most $d/2$ away from the destination, both on the left and on the right of the destination node, where d is the distance of the current node to the destination. Since in our probing procedure, we allow the usage of long-range links only if they are at the left side of $w.lrl$, the probability that at any routing step the distance to the destination would be halved is now equal to the probability that the next node of the probing path lies in $A = \{v \in \mathbb{Z}^k : dist(v,t) \geq dist(w,t)/2 \wedge id(v) < id(t)\}$.

The probability that u has a long-range link with length d decreases as d increases (as described in [4]). That means that a node in A has a higher probability to have $u's$ long-range link as a node in B.

The above fact, in combination with the fact that the number of nodes in A and B are the same, gives us:

$$Pr[lrl(u) \in A] \geq (1/2)Pr[lrl(u) \in B] \Rightarrow$$

$$Pr[lrl(u) \in A] = (1/2)\Omega\left(\frac{1}{\ln^{1+\epsilon} d}\right) \Rightarrow$$

$$Pr[lrl(u) \in A] = \Omega\left(\frac{1}{\ln^{1+\epsilon} d}\right).$$

So, by the analysis of Theorem 2 in [4], we get that the expected number of steps for a probing message to reach its destination in a stable state is $O(ln^{2+e}d)$. ■

G. Join and Leave

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

When a node joins the network, it is initially connected with an arbitrary node and it is placed to its stable position (i.e. in between its legitimate left and right neighbors) by the process of linearization. So, if u is connected initially to node

v and w.l.o.g. $u > v$, then u is forwarded to either the right next neighbor of v or possibly the long-range link of u. It is forwarded to $v.lrl$ only if $v.lrl > v$ and $v.lrl < u$. However, we can observe that these two conditions are the same as in the probing propagation procedure. We use that fact above.

Theorem 4.24: The number of steps needed to integrate a node new node u inserted in the network at a node v into its stable state position is at most $O(ln^{2+\epsilon}n)$. Also, the number of steps needed for a network to recover to its stable state after a node u leaves the network is at most $O(ln^{2+\epsilon}n)$.

Proof: We reduce the problem of finding the linearization propagation path for a node u with a link to v in a graph G to reach the stable position at the stable state to finding the probing path from a node w to $w.lrl$ in a graph G'.

We construct G' as follows. G' is the graph having the same vertices and edges as G, and we take $w.r$ as v (i.e. $w = v.l$). Also, we define the long-range link of w in G' to be $u.id$ ($u = w.lrl$). Note here that a probing message in G', starting from w through $w.r$ with destination $w.lrl$, continuously chooses as a next edge of the probing path the edge that is always closest to $w.lrl$ but at the same time being smaller than $w.lrl$. This is the same process that the linearization uses to propagate edges. In particular, in G firstly a backwards edge (v, u) would be formed and this edge would be propagated until the left neighbor of u is reached as mentioned before (as seen in Algorithm 2). So, since the linearization propagation is done between $v = w.r$ to $u = w.lrl$ and we constructed G' in such a way so that the same edges (between the nodes $w.r$ and $w.lrl$) as in G exist, the linearization propagation path in G is the same as the probing path in G' minus the edge $(w, w.r)$. That means that the lengths of these paths vary only by 1. We know from the last section that the length of the probing path is $O(ln^{2+e}dist(w, w.lrl))$. As a consequence the first part of the theorem follows.

When a node u leaves the network, it disappears from it and the connections it had to and from other nodes also disappear. As a consequence, two nodes (formerly $u.l$ and $u.r$) have no right and left neighbors respectively, and for the network to recover to its stable state, these two must connect to each other so that this "gap" will be closed. Small-world networks have been proven to be extremely robust against node failures and that fact assures us that with high probability the network will be connected after this single node failure. That means that there is at least one node v, w.l.o.g. at the left side of u, having a long-range link positioned somewhere in the right side of u. Since the edge $(u.l, u.r)$ is missing, probing is not going to succeed for the node having the smallest (in terms of length) long-range link crossing the $(u.l, u.r)$ gap. So, after at most $O(ln^{2+\epsilon}n)$ steps an edge connecting $u.l$ with $v.lrl$ is created. This edge is mirrored and propagated through linearization and (as shown above) reaches $u.r$ after another $O(ln^{2+\epsilon}n)$ steps and the theorem holds. ■

V. Conclusion

In this work, we considered a specific small-world network which was introduced by Chaintreau et. al. in [4]. Small-world networks seem to describe very well many real world systems, such as neural networks, social networks and food webs. This network uses a move-and-forget mechanism for creating links, which we adopted also for our construction. In particular, we used the one-dimensional variation of that network, placed at a ring-structured graph, and developed a randomized self-stabilization process that is able to converge to this particular small-world network from any weakly connected initial state. As a consequence, the self-stabilizing variant of this small-world network inherits also its properties, which is greedy routing in $O(ln^{2+e}n)$, as well as robustness against node failures. We proved the correctness of our self-stabilization algorithm, and we also showed that the network recovers after a single failure/insertion in $O(ln^{2+e}n)$ steps. Our paper contributes to the numerous self-stabilization methods for various graphs existing in the bibliography. This topic seems to have gained a lot of attention lately, due to the tendency of the networks to have a more dynamic nature in recent years.

For sure there is great potential for future work in the field of self-stabilizing networks. A direct extension of this paper would be, if possible, to find methods for self-stabilizing multidimensional small-world graphs. An open question is also if there exist self-stabilization processes which are less complex (less message complexity), or with less message overhead for maintaining the connectivity of the structure. Moreover, there are still interesting network types to be studied for their self-stabilizing qualities, especially, as mentioned above, in the multidimensional spectrum.

Acknowledgment

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing"(SFB 901).

References

[1] Yaneer Bar-Yam. *Dynamics of Complex Systems*. Addison-Wesley, Reading, MA, 1997.

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

[3] Jerzy Brzezinski, Michal Szychowiak, and Dariusz Wawrzyniak. Self-stabilization in distributed systems - a short survey, 2000.

[4] Augustin Chaintreau, Pierre Fraigniaud, and Emmanuelle Lebhar. Networks become navigable as nodes move and forget. *ICALP*, volume 5125 of *Lecture Notes in Computer Science*, pages 133–144, 2008.

[5] Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list. In *SSS* , pages 124–140, 2008.

[6] Curt Cramer and Thomas Fuhrmann. Self-stabilizing ring networks on connected graphs. In *Technical report*, University of Karlsruhe (TH), Fakultaet fuer Informatik, 2005-5.

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

[8] Peter Sheridan Dodds, Roby Muhamad, and Duncan J. Watts. An experimental study of search in global social networks. *Science*, 301, pages 827–829, 2003.

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

[10] Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *LATIN*, pages 294–305, 2010.

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

[12] Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC*, pages 131–140, 2009.

[13] Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. A self-stabilizing and local delaunay graph construction. In *ISAAC*, pages 771–780, 2009.

[14] Jon Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC*, pages 163–170, 2000.

[15] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: a self-stabilizing chord overlay network. In *SPAA*, pages 235–244, 2011.

[16] David Liben-Nowell, Jasmine Novak, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Geographic routing in social networks. *Proceedings of the National Academy of Sciences*, 102(33), pages 11623–11628, August 2005.

[17] Stanley Milgram. The small-world problem. *Psychology Today*, 1(1):61–67, 1967.

[18] Riza Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *SSS*, pages 356–370, 2011.

[19] Melih Onus, Andrea W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *ALENEX* pages 99–108, 2007.

[20] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

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

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

[23] Ion Stoica, Robert Morris, David Liben-nowell, David Karger, M. Frans, Kaashoek Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[24] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684), pages 440–442, 1998.

[25] Xiao Fan Wang and Guanrong Chen Complex Networks:Small-World, Scale-Free and Beyond. In *IEEE Circuits and Systems Magazine*, 2003.

[26] Shlomi Dolev and Nir Tzachar, Empire of colonies: Self-stabilizing and self-organizing distributed algorithm, In *Theor. Comput. Sci.*, pages 514-532, 2009.

[27] Shlomi Dolev and Nir Tzachar Spanders: distributed spanning expanders In *SAC*, pages 1309-1314, 2010.

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