A Self-Stabilizing Algorithm for Virtual Ring Construction

Hans-Peter Paulsen
Hamburg University of Technology
Institute of Telematics
hans.paulsen@tuhh.de

Gerry Siegemund
Hamburg University of Technology
Institute of Telematics
gerry.siegemund@tu-harburg.de

***Abstract*—This paper presents a self-stabilizing, distributed algorithm for finding a virtual ring in a connected unweighted graph, named SelfVRC. A virtual ring allows routing without knowing the topology of the underlying network. All network nodes know their own positions on the ring as well as those of their neighbors. While self-stabilizing algorithms that construct a virtual ring exist, little work has been done in minimizing its length. SelfVRC was evaluated with different fair and unfair schedulers. It stabilizes in $\mathrm{O(n)}$ rounds. The resulting ring is not longer than $\mathbf{2(n-1)}$ and is on average significantly shorter. Cycles in the underlying graph are utilized to reduce the length of the virtual ring. SelfVRC depends on unique node identifiers and a root node.**

I. Introduction

This work introduces a novel self-stabilizing algorithm to create a virtual ring. A virtual ring can be used for routing in sensor networks or as a basis for other distributed algorithms. It provides, for instance, an easy way for a token to traverse every node in sequential order. The algorithm is called Self-stabilizing algorithm for Virtual Ring Construction (SelfVRC). It finds a ring over all nodes in a system modeled by an unweighted, undirected, connected graph. Fig. 1 (a) shows a simple network with its corresponding ring in Fig. 1 (b). Neighboring nodes on the ring are also neighbors in the graph. The network is modeled as an unweighted graph $G = (V, E)$ with n nodes and m edges. Since the graph is unweighted the length of the virtual ring is defined as the number of edges in the virtual ring. A ring of minimal length that passes through every node at least once is called a Hamiltonian Walk [1], or if it includes every node exactly once, a Hamiltonian Cycle. Finding a Hamiltonian Walk is a variant of the Traveling Salesman Problem (Graph-TSP) and is, when expressed as decision problem, NP-complete. SelfVRC does not guarantee that the resulting virtual ring is the shortest possible, only that the resulting virtual ring is no longer than $2(n-1)$. A virtual ring with the length $2(n-1)$ can be achieved by constructing a spanning tree of a graph and traversing it. A self-stabilizing algorithm for this exists [2]. Removing the edge from node C to node E in Fig. 1 (a) produces a spanning tree. A spanning tree based algorithm for virtual ring construction does not make use of the cycle created by the nodes B-C-E-D. Hence, the resulting virtual ring depicted in Fig. 1 (c). It is two steps longer than the ring resulting from SelfVRC depicted in Fig. 1 (b). SelfVRC attempts to find a shorter walk by identifying cycles in the graph.

SelfVRC is designed to be self-stabilizing. Self-stabilization is a concept of fault-tolerance of distributed systems presented by Dijkstra in 1974 [3]. A self-stabilizing system will arrive at a correct state from any arbitrary start state in a finite number of execution steps. It remains in a correct state (or a set of correct states) thereafter if no fault occurs. This allows a self-stabilizing algorithm to recover from transient faults, e.g., memory errors, message loss or message corruption. It adapts to changes in network topology, e.g., the addition or removal of nodes. Self-stabilization provides non-masking fault-tolerance, i.e., it does not guarantee that the system runs correctly at all times. It guarantees that it will recover from any transient fault. Usually, nodes do not know if the algorithm is globally in a correct state. Whether an algorithm is self-stabilizing can depend on assumptions about the execution order of the nodes, which is modeled with a scheduler. The algorithm has been tested with a synchronous scheduler as well as two central schedulers one of them unfair the other fair.

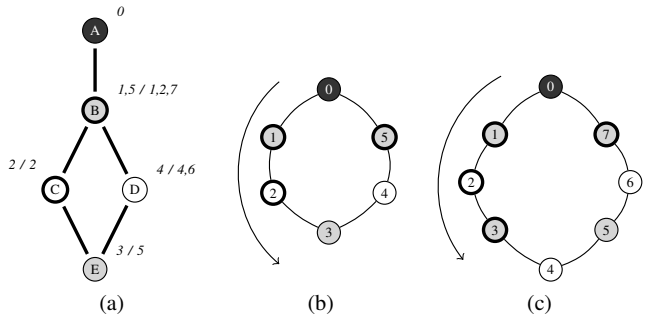

Fig. 1. (a) depicts a graph, the numbers denote the positions of its nodes on a virtual ring. The left numbers for the virtual ring depicted in (b) which is a possible result of SelfVRC. The right numbers for the virtual ring depicted in (c) which is a virtual ring resulting from constructing a spanning tree of (a) and building a virtual ring on top.

In the following we first address related work, but to the best of our knowledge there is no other self-stabilizing algorithm which attempts to minimize the length of a virtual ring. Then the algorithm is described, afterward, empirical tests are shown in relation to performance and stabilization time.

II. Related work

As mentioned in the introduction one strategy to find a closed walk is traversing a spanning tree. Self-stabilizing algorithms that create spanning trees exist [4] [5] [6]. One possible method for creating a virtual ring from a spanning tree, used by Siegemund et al. [2], works as follows: To determine the positions on the ring it recursively determines the number of nodes in the subtree rooted at each child of a node.

If a node knows its own first position and the number of nodes in the subtrees of its child nodes it can calculate its remaining positions. This results in a stabilization time proportional to the diameter of the graph.

Such a tree-based approach results in a constant walk length of $2(n-1)$. A Hamiltonian Cycle has a length of n, i.e, the result of the tree based approach is at most twice as long as the shortest possible walk.

For sufficiently dense random networks Levy et al. [7] provide an algorithm that has a high probability to find a Hamiltonian Cycle in random Graphs $G(n,p)$ if $p = \omega(\sqrt{logn}/n^{1/4})$. It terminates in $O(n)$ rounds, whereas the expected number of rounds is $O(n3/4+\epsilon)$. A Hamiltonian Cycle forms the shortest possible ring. However, the existence of a Hamiltonian Cycle is not guaranteed and if one exists it is not certain that it is detected by the algorithm. Additionally, the algorithm requires a synchronous scheduler and is distributed but not self-stabilizing.

III. Algorithm

SelfVRC finds a virtual ring in a graph. To reduce the length of the virtual ring it identifies cycles in the underlying graph. It depends on unique identifiers (ID), a root node and a connected graph. If no root node exists a leader election algorithm can be utilized to select one. SelfVRC uses the shared memory model [8], i.e., the variables mentioned in this section can be read by all neighbors of a node but only the node itself can write to the variable. SelfVRC is partitioned into three algorithms.

1) $\mathcal{D}$is algorithm: Determines distance to root.
2) $\mathcal{C}$ycle algorithm: Marks cycles in the graph.
3) $\mathcal{P}$os algorithm: Assigns positions to create the virtual ring.

SelfVRC is the collateral composition [8] of these three algorithms. The algorithms are ordered $\mathcal{D}$is, $\mathcal{C}$ycle, and $\mathcal{P}$os from lowest to highest. Algorithms of lower order are independent of algorithms with higher order. While an algorithm with lower order has not finished arbitrary output is created by algorithms with higher order. Since all three algorithms are self-stabilizing they will recover once the algorithms with lower order have stabilized. In the following it will be assumed that algorithms with lower order are already stable when discussing algorithms with higher order.

A. $\mathcal{D}$is algorithm

The $\mathcal{D}$is algorithm determines the distance to the root node. The root node has the distance 0, while all other nodes pick the minimum distance among their neighbors and increment it by one. The procedure is similar to the creation of a spanning tree.

B. $\mathcal{C}$ycle algorithm

The $\mathcal{C}$ycle algorithm finds and marks cycles in the graph. The identification of cycles is based on the following observation: Every cycle contains at least one node v with two neighbors on the cycle whose distance is not greater than the distance of v. This node is referred to as *origin-node*. The $\mathcal{C}$ycle algorithm identifies, for nodes on a cycle, the nodes prior to them on the cycle (predecessor) or those that come after (successor). Every node has two variables, *S* and *P*, and the IDs of successor and predecessor are assigned to these variables. As mentioned before a shared memory model is used, i.e., nodes share the variables *S*, *P*, and cycle-ID (CID) of

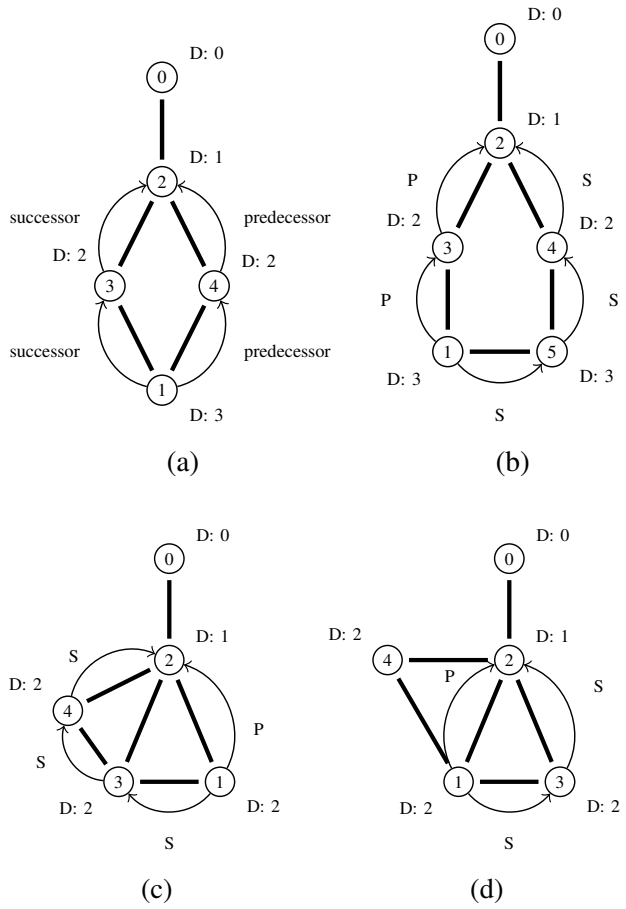

Fig. 2. Propagation of cycles. *D* denotes the distance to the root node.

their neighbors. The *origin-node* takes its own ID as CID. The $\mathcal{C}$ycle algorithm marks nodes as part of a cycle by assigning cycle-IDs, nodes with the same CID are considered part of the same cycle.

The cycle propagates because nodes become part of the cycle if the variables *S* or *P* of a neighbor match their IDs. Those nodes adopt the same CID and choose a predecessor/successor in turn. This way the CID propagates until the end points of the cycle meet. The node where the cycle closes is referred to as *connecting-node*.

The cycles found by the $\mathcal{C}$ycle algorithm overlap in at most one node, since using two cycles that overlap in more than one node is not guaranteed to lower the length of the virtual ring and might increase it instead. Limiting overlap means that the $\mathcal{C}$ycle algorithm can not complete some cycles, incomplete cycles are referred to as *cycle-fragments*.

If a node has a neighbor with the same distance, then the cycle could begin spreading from either node. In such a case the node with the lower ID chooses the node with the higher ID as successor.

- A node with the same distance can be chosen as successor if it has higher ID and does not already have a lower *CID*.
- Nodes with the same distance are chosen over nodes with a lower distance for the successor.
- The predecessor is chosen from among nodes with a lower distance.

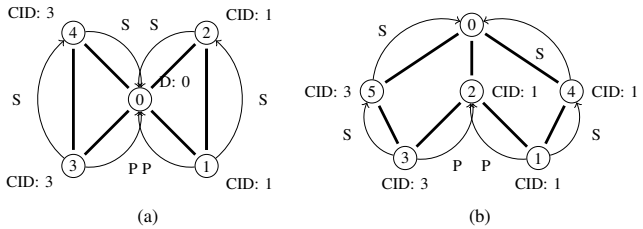

Fig. 3. Shows overlapping cycles, in (a) both are complete because origin-nodes can be shared. In (b) the left cycle is incomplete since node 2 belongs to the right cycle which has the lower CID. S denotes successor, P predecessor.

- For symmetry breaking the lower ID is always chosen over the higher ID.

That the choice of predecessor and successor nodes is restricted to nodes with lower distance or the same distance and a higher ID prevents loops. Loops refer to situations where there is no *origin-node* but, e.g., a node v is part of the cycle because node w chose it as successor while w is part of the cycle because v has chosen it as successor. With the given conditions if v is a predecessor/successor candidate for w, then w can not fulfill the conditions to be a candidate for v. That means there has to be an *origin-node*. Furthermore, it ensures that the end points meet, if not interrupted by another cycle. There are only a limited number of nodes with the same distance that can be chosen at every distance and only a limited number of times a node with lower distance can be chosen before reaching the root node. Therefore, if the ends of the cycle do not meet before reaching the root node they will meet at the root node. Due to space limitations we omit certain special cases like the former.

Fig. 2 shows examples for the propagation of cycles. In Fig. 2 (a) the neighbors of node 1 both have a distance of 2. The *connecting-node* is node 2 the *origin-node* is node 1. Fig. 2 (b) shows two neighboring nodes, 1 and 5, with the same distance, node 1 chooses node 5 as successor and the CID is 1, because 1 is the node with the lower ID. In Fig. 2 (c) the nodes 1, 3, and 4 have the same distance. Node 1 chooses node 3 as successor which chooses node 4 as successor. In Fig. 2 (d) node 1 is between node 3 and 4 instead. Node 1 still chooses node 3 as successor but node 4 is not part of the cycle. This shows how the results depend on the distribution of IDs.

A cycle can interrupt another cycle because each node can only have one cycle-ID. The *connecting-node* of a cycle does not adopt the CID of the cycle which allows cycles to share the same *connecting-node* or for the *connecting-node* of one cycle to be part of another cycle. Nevertheless, every node can only have one CID. If a node has been chosen as successor/predecessor of more than one node with different CIDs, it joins the cycle with the lowest CID.

In Fig. 3 (a) the cycle consisting of the nodes 0, 1, 2, and the cycle consisting of the nodes 0, 3, 4 share 0 as *connecting-node* and both cycles are complete. However, in Fig. 3 (b) node 1 and node 3 choose node 2 as predecessor and node 2 takes 1 as its CID because the cycle with the lower CID has precedence. Which leaves the cycle with CID 3 incomplete, i.e., a *cycle-fragment*.

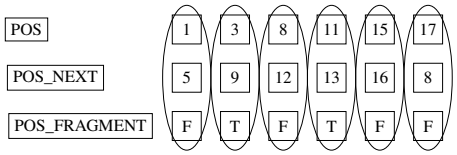

Fig. 4. The three lists of the $\mathcal{P}$os algorithm have the same number of elements.

C. $\mathcal{P}$os algorithm

The $\mathcal{P}$os algorithm builds the ring order based on the other two algorithms by determining which neighboring nodes come before or after a node on the virtual ring.

- Nodes not marked as part of a cycle, i.e., those with an empty *CID*, come after the neighbor which has a lower distance. Nodes without a CID only have one neighbor with a lower distance. A node with two neighbors with a lower distance becomes an *origin-node* if it does not have a CID already.
- The first position of nodes on a cycle is the position after the last position of their predecessor. Complete cycles are traversed starting from the *connecting-node*.
- *Cycle-fragments* do not have a *connecting-node*. *Cycle-fragments* are divided into two parts, in one part the first position of the nodes is the position succeeding one of the positions of their predecessor for the other it is their successor. (The reason for this will not be explained in this paper due to space limitations.)

The position algorithm uses three lists, *POS*, *POS_NEXT*, and *POS_FRAGMENT*. These lists have the same amount of elements and the i-th element of each list corresponds to the i-th element of the other two lists, see Fig. 4. *POS* contains a node's positions. The corresponding elements of the *POS_NEXT* list contain the IDs of the neighbors which have the next position on the ring, i.e., those which fit into one of the three categories in the list above. The *POS_FRAGMENT* list is a list of booleans. An entry is set to true to mark a neighbor as part of a *cycle-fragment*.

The root node begins propagating the positions along the ring. A node knows from which node it will get a position. It searches for its own ID in the *POS_NEXT* list of the neighbor it expects a position from. If there is a matching entry in the *POS* list it increments the position and chooses the next node in turn.

The nodes in a *cycle-fragment* do not have enough information to determine whether the nodes with the same *CID* form a complete cycle. However, if a node v chose a node w as successor or predecessor and w stopped the propagation of the cycle by adopting another *CID*, then w knows v is part of a *cycle-fragment*. Node v gets its first position from w and the entry in the *POS_FRAGMENT* list corresponding to its ID in the *POS_NEXT* list is set to *TRUE* to mark a *cycle-fragment*. The information propagates together with the position, since nodes on a *cycle-fragment* mark the next node in turn.

After the $\mathcal{P}$os algorithm stabilizes every node knows the positions of itself and its neighbors. For every position there is a neighboring node with the previous or the next position on the ring.

IV. Empirical tests - Setup

To collect statistical data about the resulting ring as well as stabilization times the algorithm has been simulated in python with randomly generated graphs and start states. The execution order of the nodes is determined by three different schedulers: a fair central scheduler, an unfair central scheduler and a synchronous scheduler. The main graph generation model used is the well known Erdős-Rényi (ER) model. About 11 million Graphs created with the ER model were tested, most of these had between 5 and 170 nodes. The largest tested graphs were 33000 graphs with 1000 nodes.

A. Schedulers

Nodes evaluate their rules separately from each other but depend on the state of neighboring nodes. Thus, it is important in which order the nodes execute their rules since the results of node x executing before node y can be different than the opposite or node x executing at the same time as node y. E.g., if two nodes have either the state *false* or *true*, follow a rule to adopt a state different than the state of their neighbor, and both begin as *true*. Then the behavior is different for a scheduler where nodes evaluate their rules at the same time and a scheduler where they do not. If they execute at the same time both will alternate between *true* and *false* each turn and the system does not stabilize. If one node executes first it will switch to false and the system is in a stable state.

Three different schedulers have been used:

1) A synchronous scheduler where in each round all nodes that can make a move do so at the same time. Which means all nodes work based on the state their neighbors had at the end of the last round.
2) A fair central scheduler. With a central scheduler only one node executes at a time. It is fair because in every round every node that can make a move makes exactly one move. The execution order in a round is random. In the following this scheduler is referred to as fair scheduler, although a synchronous scheduler is also fair.
3) An unfair central scheduler, which in the following is referred to as random scheduler. The execution order is completely random, a node can make an arbitrary amount of moves before another makes its first move.

B. Graph generation models

1) Erdős-Rényi model: Random testing requires a model for generating random graphs, the different models are used to test for difference with graphs with different properties. The main model we use is the $G(n, p)$ variant of the ER model. The ER model does not guarantee a connected graph although it is likely if $p > (ln(n)/n)$ [9]. Since the algorithm assumes the graph is connected a new graph is generated if an unconnected graph was generated, the same applies for the Wattz and Strogatz model.

2) Wattz and Strogatz model: The Wattz and Strogatz model [10] uses 3 parameters: n - the number of nodes, the mean degree K (which is an even integer) and the rewiring probability p.

It creates n nodes arranged in a circle. Each node is connected with its K nearest neighbors. Every edge has a chance of p to be rewired, i.e., an edge between nodes i and j is replaced with an edge between the node i and a random node k. With $p = 0$ the result is a regular ring lattice, while with growing p the Wattz and Strogatz model approaches the ER model.

3) Barabási-Albert model: The Barabási-Albert model [11] generates scale-free networks by using preferential attachment. A scale-free network is one where the degree distribution follows a power-law and it is another property ER graphs do not possess. Graphs created with the Barabási-Albert model tend to have hubs connected to many nodes.

The model takes two positive integers as parameters, M and n with $M < n$. M nodes without edges are created, then nodes with M edges to existing nodes are added until there are n nodes. Since the first node added has to have an edge to all other M nodes and every node added has edges to existing nodes the result is guaranteed to be connected. Also, new nodes are more likely to connect to nodes which have a high degree. The chance that a node is connected to an existing node is their degree divided by the sum of the degrees of all existing nodes.

Since the algorithm depends on the IDs of the nodes and the IDs are assigned in ascending order around the ring, the IDs are randomly reassigned before the graph is used, the same applies for the Wattz and Strogatz model.

C. Setup

The test procedure is as follows: A graph is created with one of the models. The initial state of the algorithm is randomly created. If all possible states were tested that would also include all states that could be caused by transient errors. There are too many possible states for an exhaustive test, nonetheless testing a huge number of random initial states also indicates that SelfVRC can recover from faults.

The result is checked by following the virtual ring starting from the root node in the order of the positions assigned by the $\mathcal{P}$os algorithm. The virtual ring has to contain all nodes, end at the root node and cannot be longer than $2(n-1)$. Otherwise, an error is reported.

SelfVRC performs without errors. About 11 million ER graphs were tested as well as 240000 Barabási-Albert graphs and 24000 Wattz and Strogatz graphs.

For ER graphs - each repeated for each scheduler:

- For $G(n, 20\%)$ and $G(n, 40\%)$ with $n = 5, ..., 170$, 10000 trials were performed with each graph size and probability.
- For $G(n, 20\%)$ and $G(n, 40\%)$ with $n = 171, ..., 300$, 1000 trials were performed with each graph size and probability.
- For $G(100, p)$ with $p = 4\%, 8\%, ..., 92\%, 96\%$, 10000 trials were performed with each probability.
- For $G(1000, p)$ 33000 trials were performed with probabilities between 1% and 13%.

Barabási-Albert graphs were only tested with the fair scheduler and 100 nodes, with $M = 4, 8, ..., 92, 96$. 10000 graphs were tested per value of M. Wattz and Strogatz graphs were only tested with the fair scheduler and 100 nodes, with $p = 10\%$ and $K = 4, 8, ..., 92, 96$. 10000 graphs were tested per value of K.

V. Empirical tests - Results

1) Length: Fig. 5 shows how the length of the resulting virtual ring is distributed for ER graphs with 100 nodes for different probabilities p. The length is based on the number of edges in the virtual ring, the minimum ring length is n, 100 in

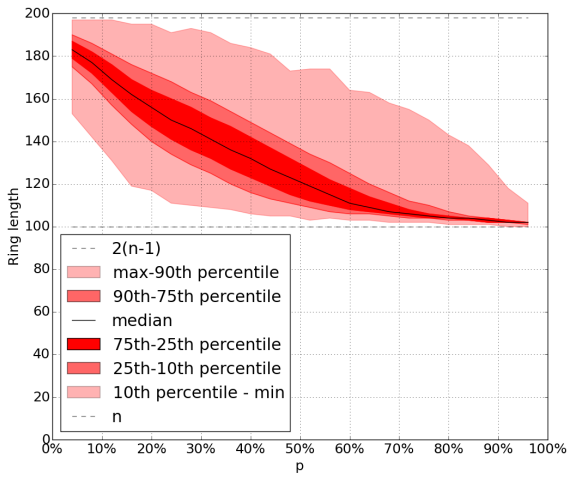

Fig. 5. Ring length with different probabilities and 100 nodes in the ER model $G(100, p)$.

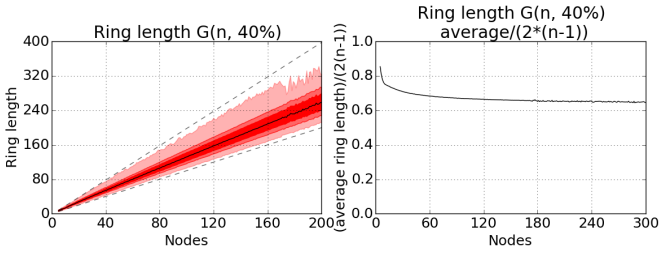

Fig. 6. Ring length in relation to number of nodes with $G(n, 40\%)$. Right side average ring length divided by $2(n-1)$. With the ER model.

this case. Furthermore, the goal was an algorithm which does not exceed the length of a virtual ring based on the traversal of a spanning tree which is $2(n-1)$ or 198 hops in this case.

As Fig. 5 shows the ring length has a correlation to p, and, while the minimum and maximum values vary strongly, 80% of the results can be found in a region around the median which at its widest extension spans a range of fewer than 40 hops. The median falls monotonously with growing p because on average denser graphs contain a higher number of cycles. For $p > 65\%$ less than 10% of the resulting virtual rings have more than 120 hops.

Fig. 6 shows how the ring length varies with the number of nodes. Up to 170 nodes the charts are based on at least 30000 data points per number of nodes, while from 171-300 nodes they are only based on 1000 tested graphs per graph size. For this reason they are less smooth after this point.

Fig. 6 (right) shows the average length divided by $2(n-1)$. With growing graph size a clear decline of this value is visible. However, the trials with 1000 nodes indicate that the influence of this trend is limited. For $G(100, 12\%)$ the average length is about 169. While for $G(1000, 12\%)$ it is 1601.

As for Wattz and Strogatz graphs, they produce shorter virtual rings with a comparable number of edges as can be seen in Fig. 7. How the result changes with different rewiring probabilities has not been tested. With 100 nodes a Wattz and Strogatz Graph has $100K/2$ edges whereas an ER graph has on average $100(99 \cdot p)/2$ edges, i.e., with $K = x$ and $p = x\%$ the ER graph has on average 1% fewer edges than the Wattz and

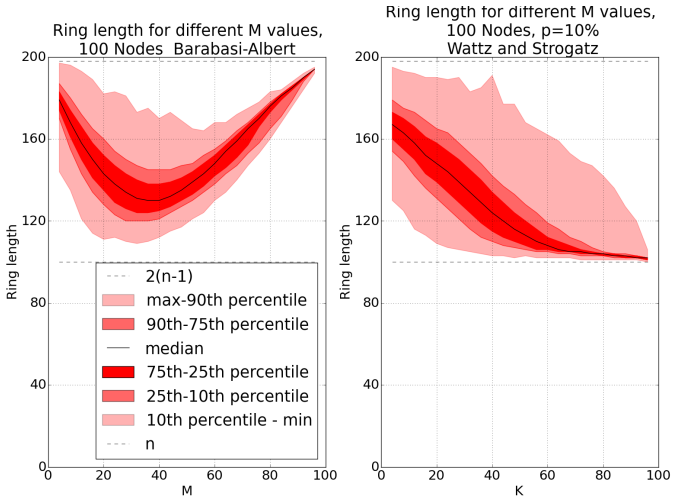

Fig. 7. Ring length in relation M/K for Barabási-Albert on the left and Wattz and Strogatz on the right with 100 nodes.

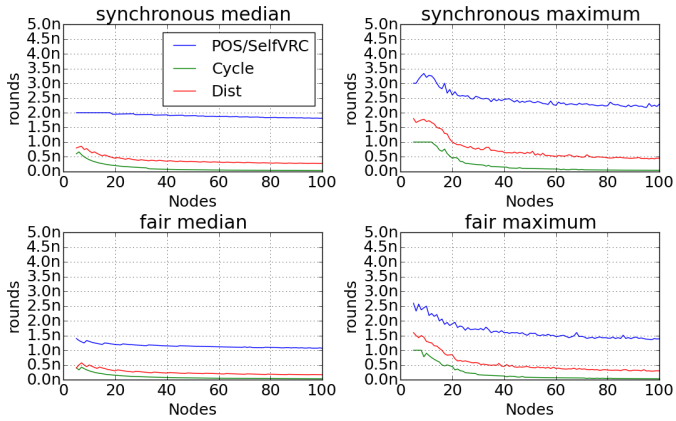

Fig. 8. Stabilization time in rounds as multiple of n and as function of p, with fair and synchronous scheduler. Green for the $\mathcal{D}$ist algorithm, red for the $\mathcal{C}$ycle algorithm, blue for the $\mathcal{P}$os algorithm/complete algorithm. The value for the $\mathcal{P}$os algorithm includes the rounds running parallel with the other two, and the rounds for the $\mathcal{C}$ycle include the time running parallel with the $\mathcal{D}$ist algorithm. ER model $G(n, 40\%)$.

Strogatz graphs which makes the graph comparable. With $K = 4$ and 200 edges the median length is 167 whereas ER graphs with $p = 4\%$, which have on average 198 edges, produce a median length of 184. The reason is most likely that Wattz and Strogatz graphs are based on a ring lattice. With $K = 2$ it starts as a ring containing all nodes and with growing K more connections to nearby nodes are added. It is likely that with p=10% rewiring forms smaller cycles instead of creating a tree-like structure. A higher number of cycles makes it more likely that the algorithm finds one.

Fig. 7 also shows how Barabási-Albert behaves with different M values. One reason the length first falls and then begins to increase again is that Barabási-Albert graphs with 100 nodes have $(100-M)M$ edges. The maximum amount is 2500 edges with $M = 50$. At $M = 99$ a single node is connected to all others without any other connections. However, the lowest point of the median is about $M = 36$ not $M = 50$. The reason might be an increasing hub size which decreases the number of connections between non-hub nodes, which makes long cycles less likely.

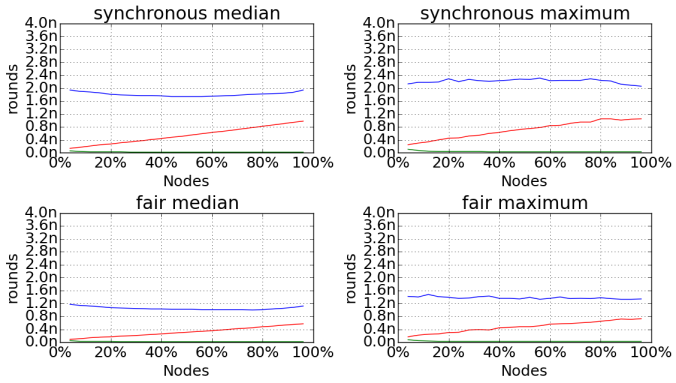

Fig. 9. Stabilization time in rounds as multiple of n and as function of the number of nodes, with fair and synchronous scheduler. Green for the $\mathcal{D}$ist algorithm, red for $\mathcal{C}$ycle algorithm, blue for the complete algorithm. ER model $G(100, p)$.

2) Stabilization Time - Rounds: Fig. 8 shows how the stabilization time in rounds changes with a growing number of nodes. The random scheduler has longer but fewer rounds with more moves and for that reason gives no further insight in this section. The synchronous scheduler results in a higher number of rounds than the fair scheduler. With the central fair scheduler nodes can take moves earlier in the same round into account. The $\mathcal{P}$os algorithm tends to need the highest amount of rounds to stabilize and with synchronous rounds the position can only propagate by one node per round. Whereas with the central fair scheduler there is a 50% chance that it advances by two nodes in a round and a lower chance that it advances more than two.

With a growing number of nodes the stabilization time of the $\mathcal{D}$ist algorithm becomes negligible. The reason is, that in ER graphs the average path length does not grow linearly with the number of nodes, the diameter is proportional to $log(N)/log(Np)$. The $\mathcal{D}$ist algorithm never needs more than n rounds. The whole algorithm never needed more than $3.2n$ rounds. With $p = 40\%$ the median stabilization time for the fair scheduler settles at n rounds and for the synchronous scheduler $1.7n$ rounds.

Varying p does not change the stabilization time in rounds by more than $0.4n$, as seen in Fig. 8. While the stabilization time of the $\mathcal{C}$ycle algorithm increases with p the stabilization time of the $\mathcal{P}$os algorithm decreases because of shorter ring lengths. A possible explanation for the increase in the stabilization time of the $\mathcal{C}$ycle algorithm is that a denser network makes it more likely that the algorithm can find bigger cycles which need more rounds to stabilize than many smaller cycles expanding in parallel.

3) Stabilization Time - Moves: The unfair variant of the central scheduler produces a higher number of moves than the fair variant because a single round can consist of a higher amount of moves. Fig. 10 shows that although the number of moves with the random scheduler could theoretically be very high, in practice it needs fewer moves than the synchronous schedules and the number of moves varies by less than $0.02n^2$ for graphs with more than 60 nodes. While the theoretical worst case number of moves is high in practice the number of moves seems to grow proportionally with n^2. However, that only indicates that the worst case is relatively unlikely. The increase in the number of moves with p can be linked to the slower stabilization of the $\mathcal{C}$ycle algorithm.

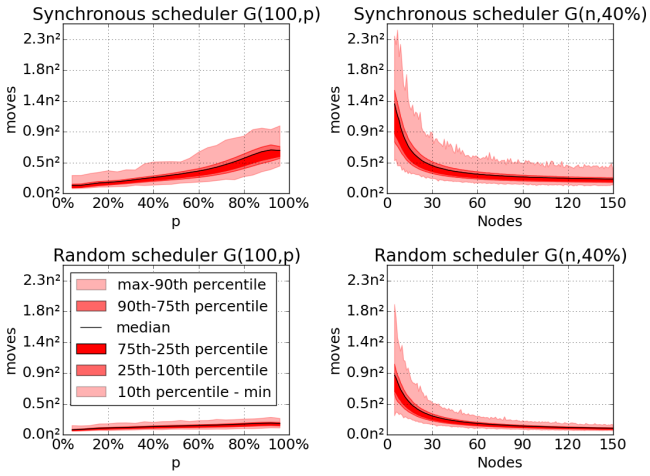

Fig. 10. Stabilization time in moves as multiple of n^2 with synchronous and random scheduler. On the left with 100 nodes as function of p. On the right with $p = 40\%$ as function of n. ER model $G(100, p)$ and $G(n, 40\%)$.

VI. Conclusion

The paper introduces a novel algorithm for virtual ring creation. Based on experimental data it stabilizes in $O(n)$ rounds, although there is no formal proof. It is the collateral composition of three algorithms and has been tested with a synchronous scheduler, a fair central, and an unfair central scheduler.

In comparison to generating a virtual ring from a spanning tree SelfVRC's stabilization time scales with the number of nodes in the graph instead of its diameter and it requires more memory. In exchange the algorithm offers significant improvements in virtual ring length, especially in dense networks.

References

[1] T. Nishizeki, T. Asano, and T. Watanabe, "An approximation algorithm for the hamiltonian walk problem on maximal planar graphs," *Discrete Applied Mathematics*, vol. 5, no. 2, pp. 211 – 222, 1983.

[2] G. Siegemund, V. Turau, and K. Maâmra, "A self-stabilizing publish/subscribe middleware for wireless sensor networks," in *Proceedings of the International Conference on Networked Systems (NetSys)*, Mar. 2015, pp. 1–8.

[3] E. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, 1974.

[4] A. Kosowski and Ł. Kuszner., "A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves," in *Parallel Processing and Applied Mathematics*. Springer, 2006, pp. 75–82.

[5] S. Huang and N.-S. Chen, "A self-stabilizing algorithm for constructing breadth-first trees," *Information Processing Letters*, vol. 41, no. 2, pp. 109–117, 1992.

[6] N.-S. Chen, H.-P. Yu, and S.-T. Huang, "A self-stabilizing algorithm for constructing spanning trees," *Information Processing Letters*, vol. 39, no. 3, pp. 147 – 151, 1991.

[7] E. Levy, G. Louchard, and J. Petit, "A distributed algorithm to find hamiltonian cycles in $\mathcal{G}$(n, p) random graphs," in *Combinatorial and Algorithmic Aspects of Networking*, ser. Lecture Notes in Computer Science, A. López-Ortiz and A. Hamel, Eds. Springer Berlin Heidelberg, 2005, vol. 3405, pp. 63–74.

[8] S. Dolev, *Self-stabilization*. Cambridge, Mass, London, England: MIT Press, 2000. [Online]. Available: http://opac.inria.fr/record=b1096214

[9] P. Erdős and A. Rényi, "On random graphs. i," *Publicationes Mathematicae*, 1959.

[10] D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, 1998.

[11] A. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, 1999.