Uniform Deterministic Self-Stabilizing Ring-Orientation on Odd-Length Rings

J-H. Hoepman

Computer Science/Department of Algorithms and Architecture

CS-R9423 1994

Uniform Deterministic Self-Stabilizing Ring-Orientation On Odd-Length Rings

Jaap-Henk Hoepman
jhh@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

The ring-orientation problem requires all processors on an anonymous ring to reach agreement on a direction along the ring. A self-stabilizing ring-orientation protocol eventually ensures that all processors on the ring agree on a direction, regardless of the initial states of the processors on which the protocol is started. In this paper we present two uniform deterministic self-stabilizing ring-orientation protocols for rings with an odd number of processors using only a constant number of states per processor. The first protocol is an adaption of a randomized protocol presented by Israeli and Jalfon [IJ93], and operates in the link-register model under the distributed daemon. The second protocol operates in the state-reading model under the central daemon, and complements the impossibility results proven in [IJ93]. Both protocols do not assume an upper-bound on the length of the ring, and are therefore applicable to dynamic rings. Applying our results we are able to prove that on an odd-length ring, the link-register model and the state-reading model are, in some sense, equivalent.

AMS Subject Classification (1991): 68M10, 68Q22, 90B12
CR Subject Classification (1991): C.2.4, C.4, D.1.3, D.4.1, D.4.4, D.4.5
Keywords & Phrases: Rings, Orientation, Self-Stabilization, Communication, Synchronization, Shared Memory, Symmetry Breaking
Note: Partially supported by the Dutch foundation for scientific research (NWO) through NFI Project ALADDIN, under contract number NF 62-376

1. Introduction

On oriented rings, processors agree on a direction along the ring. Distributed algorithms on rings are more easily derived if it is known that the ring is oriented (cf. [ASW88]), and may be more efficient than equivalent algorithms for unoriented rings (cf. [San84]). To orient a ring the processors must choose a left and right neighbour consistently around the ring, such that the left neighbour of each processor considers that processor to be its right neighbour. Processors can distinguish between a first and second neighbour, but to exclude trivial solutions the processors are required to be otherwise identical.

A comprehensive study on uniform rings in the asynchronous message passing model has been published by Attiya et. al. [ASW88]. They show that there is no deterministic protocol to orient even-length rings, and that there also cannot exist a protocol to orient rings of arbitrary length, if the protocols are required to terminate. Syrotiuk and Pachl [SP87] present a simple asynchronous ring-orientation protocol using message passing, that is only guaranteed to work for rings whose length is odd and bounded. Neither paper addresses self-stabilization.

A self-stabilizing protocol is a protocol that will eventually satisfy its specification, regardless of the initial state it was started in. Self-stabilization was introduced by Dijkstra [Dij74, Dij82], and is an important framework in which one can derive fault-tolerant protocols capable of recovering from transient errors. This type of error can change the state of certain processors, but leaves the processors themselves in working order. Now consider a self-stabilizing protocol running on a set of processors, and consider the state just after the last error. This could just as well have been the initial state of the protocol, so the protocol must attempt to recover from this error. As the protocol is self-stabilizing it will be able to do so, provided the next error does not occur too soon. Therefore if transient errors are infrequent enough, self-stabilizing protocols keep the system in a correct state most of the time.

Our interest in self-stabilizing ring-orientation protocols is twofold. First of all, Burns and Pachl [BP89] have shown that deterministic self-stabilizing protocols can break symmetry on oriented rings of prime size. It is natural to ask whether the ring has to be oriented to achieve this. Secondly, if a ring can be oriented deterministically, we are interested in the necessary cost of doing so.

Several models for inter-processor communication and processor scheduling in self-stabilizing systems have been proposed in the literature. In the *state-reading* model [Dij74] processors communicate by reading each others state. Secondly, in the *link-register* model [DIM93] two processors communicate using two registers, each processor reading the register written by the other. Thirdly, in the *message-passing* model processors communicate by sending messages (a)synchronously to other processors, where each message may incur an unbounded but finite delay.

The scheduling of processor steps is also an important issue. The *central daemon* [Dij74] schedules one processor at a time, which then performs one step in which it reads the information it needs, and writes its new state before returning control to the daemon. The *distributed daemon* [Bur87] may schedule several processors concurrently, but it is assumed that all scheduled processors first read the information they need, before any of them is allowed to write. Finally, the *read/write daemon* [DIM93] allows arbitrary interleaving of processor steps, much like the interleaving semantics considered for wait-free shared memory constructions.

Self-stabilizing ring-orientation protocols were studied by Israeli and Jalfon [IJ93]. They prove that no uniform deterministic self-stabilizing ring-orientation protocols exist in (a) the link-register model under the distributed daemon for even-length rings, (b) the state-reading model for either (b1) even-length rings under the central daemon, or (b2) odd-length rings under the distributed daemon. This leads them to the construction of a randomized self-stabilizing ring-orientation algorithm in the link-register model under the distributed daemon for arbitrary rings. To complement their impossibility results, they also present a uniform deterministic self-stabilizing ring-orientation protocol to orient odd-length rings in the link-register model under the distributed daemon. This protocol assumes knowledge of an upper bound on the length of the ring, and thus uses a non-constant number of states per processor.

We present a uniform deterministic self-stabilizing ring-orientation protocol in the link-register model under the distributed daemon for odd-length rings using only a constant number of states per processor. Our protocol is an adaption of the general randomized Israeli-Jalfon protocol. Contrary to the deterministic Israeli-Jalfon protocol for odd-length rings, our protocol does not depend on the length of the ring. This implies that our protocol can be used on dynamic rings on which the number of processors may change over time (provided that the length of the ring is odd in between these changes). Complementing the impossibility results of Israeli and Jalfon, we also present a uniform deterministic self-stabilizing ring-orientation in the state-reading model under the central daemon for odd-length rings using only a constant number of states per processor. Note that these results do not contradict the impossibility results of Attiya et. al. [ASW88], as self-stabilizing protocols can never be required to terminate. Neither do these results contradict the impossibility of uniform self-stabilizing mutual-exclusion on rings of non-prime length proven by Dijkstra [Dij82], as a cyclic symmetrical configuration is not necessarily unoriented

The structure of this paper is as follows. Section 2 describes the model of a distributed system assumed throughout this paper. Several formal definitions of self-stabilization and their ramifications are discussed in Section 3. Then in Section 4 we start with a brief description of the Israeli-Jalfon protocol and continue presenting our uniform deterministic self-stabilizing ring-orientation protocol in the link-register model under the distributed daemon for odd-length rings. Section 5 describes the uniform deterministic self-stabilizing ring-orientation protocol in the state-reading model under the central daemon for odd-length rings. We conclude this paper showing the equivalence of the link-register and the state-reading model on odd-length rings in Section 6, and presenting some interesting directions of further research in Section 7.

2. The Model

We consider an *anonymous ring* $R = (V_R, E_R)$, consisting of n clockwise numbered *nodes* $q \in V_R = \{0, \dots, n-1\}$ and *clockwise edges* $e \in E_R = \{pq \mid p, q \in V_R \wedge q = (p+1) \bmod n\}$. The nodes of the ring are numbered purely for notational convenience: as the ring is anonymous no node has access to its number. Two nodes p, q are *neighbours* if either pq or qp is an element of E_R. Neighbouring nodes can communicate with each other directly. Each node can distinguish a first and second neighbour. For neighbours p of q we define $rank_q(p)$ to equal 1 if p is the first neighbour of q, and 2 if p is the second neighbour of q[1].

In this paper two models of communication are considered. In the *link-register* model, a node q will communicate with its neighbours p and r using separate registers: R_{qp} is written by q and read by p, whereas R_{qr} is written by q and read by r (see Figure 1). In the *state-reading* model q stores its state in a register R_q readable by both its neighbours p and r.

The *state* $s_q \in \mathcal{S}_q$ of a node q is comprised of its internal state and the contents of the registers it writes. The *configuration* $c \in \mathcal{C}$ of the system is the Cartesian product $\prod_{q \in V_R} s_q$ over the states of all nodes in the ring. We write $c.s_q$ for the state of node q in configuration c, and similarly $c.r_q$ for the value of register R_q in configuration c. Node q can update its state according to its *program* δ_q. Each *step* of node q in configuration c changes q's state to $\delta_q(c)$. In the link-register model, $\delta_q(c)$ is defined as $\delta_q(c.r_{pq}, c.s_q, c.r_{rq})$. In the state-reading

[1] Of course node q is not necessarily the first neighbour of node p if $pq \in E_R$.

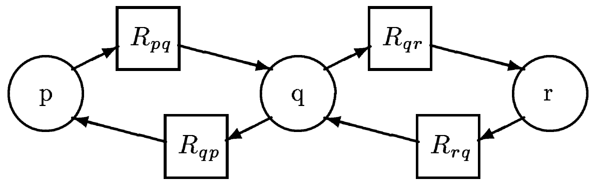

Figure 1: Registers used by q and its neighbours to communicate with each other in the link-register model

model, $\delta_q(c)$ is defined as $\delta_q(c.r_p, c.s_q, c.r_r)$. A *protocol* consists of a program δ_q for every node $q \in V_R$. A protocol is *uniform* if for all $p, q \in V_R$ we have $\delta_p = \delta_q$.

A *schedule* is an infinite sequence $(P_i)_{i\geq 0}$ of *activations* $P_i \subseteq V_R$. A schedule is *fair* if each node $q \in V_R$ occurs in infinitely many activations P_i. Define the sequence $(t_i)_{i\geq 0}$ for a given schedule $(P_i)_{i\geq 0}$ by requiring $t_0 = 0$ and, for all $j > 0$, setting t_j to the minimal i such that $(\forall q \in V_R, \exists k : t_{j-1} \leq k < i \wedge q \in P_k)$. This sequence is unique, and partitions the schedule into *rounds* i, starting at t_i and ending at t_{i+1}, such that in each round each node is activated at least once. Thus a fair schedule is partitioned into infinitely many rounds. Under the *central daemon* one node is activated at a time to execute exactly one step: hence the central daemon only *allows* P_i that consist of exactly one node. Under the *distributed daemon* a set of nodes is simultaneously activated to concurrently execute exactly one step each. It allows arbitrary P_i. All nodes executing a step must have read the values in neighbouring registers before any node can be allowed to write the new value. Under both daemons activating P_i in configuration c yields a configuration c', denoted $c \rightarrow_{P_i} c'$, such that for all $q \notin P_i$ we have $c'.s_q = c.s_q$, whereas for all $q \in P_i$ we have $c'.s_q = \delta_q(c)$.

A schedule $(P_i)_{i\geq 0}$ and an *initial configuration* $c_0 \in \mathcal{C}$ induce an *execution* $E = (c_i)_{i\geq 0}$ such that for all $i \geq 0$ we have $c_i \rightarrow_{P_i} c_{i+1}$. We write c_i^E for the i-th configuration in execution E, and if $i \leq j$ we write $c_i^E \Rightarrow_E c_j^E$. An execution is fair if it is induced by a fair schedule[2]. We write $\mathcal{E}$ for the set of all fair executions allowed by the daemon under consideration. Let the schedule be partitioned into rounds as above. Then in execution E, round i starts in configuration $c_{t_i}^E$, for which we write C_i.

We use some additional definitions in this paper. A property X is called *stable* in configuration c (of execution E) if X holds in all configurations c' with $c \Rightarrow_E c'$. A property X is called stable from configuration c_f up to configuration c_l (of execution E) if X holds in all configurations c' with $c_f \Rightarrow_E c' \Rightarrow_E c_l$. A *clockwise chain* is a sequence of nodes $q_0 \ldots q_k$, not necessarily $k < n$, such that for all i with $0 \leq i < k$, $q_i q_{i+1} \in E_R$. An *anti-clockwise chain* is a sequence of nodes $q_0 \ldots q_k$ such that for all i with $0 \leq i < k$, $q_{i+1} q_i \in E_R$. A *chain* is either a clockwise or an anti-clockwise chain.

[2]Note that we do allow stuttering (i.e. transitions $c \rightarrow_P c$) to occur in executions. In fact we need stuttering to make all executions infinite.

3. About self-stabilization

A self-stabilizing protocol is a protocol that, when started in an arbitrary initial configuration, will eventually behave according to its specification. If we want to give a formal definition of self-stabilization, we first have to formalize what we mean by the specification of a protocol. In the early papers on self-stabilization the specification was viewed as describing the set of configurations $\mathcal{L} \subseteq \mathcal{C}$, called the *legitimate configurations*, the protocol should be in. A mutual exclusion protocol, for instance, should always be in a configuration in which at most one node is executing its critical section. In this setting a protocol is *self-stabilizing* to a specification $\mathcal{L}$ if for every execution E the protocol will eventually reach a legitimate configuration $c_i^E \in \mathcal{L}$, and once the configuration is legitimate it will remain legitimate forever. I.e. if $(\forall E \in \mathcal{E}, \exists i \geq 0 : c_i^E \in \mathcal{L})$ and $(\forall c \in \mathcal{L}, \forall P : c \rightarrow_P c' \Rightarrow c' \in \mathcal{L})$.

One drawback of configuration-based specifications becomes apparent if we look at mutual exclusion. Usually it is required that the privilege, i.e. which node is allowed to execute its critical section, is passed fairly among all nodes competing. This fairness-property cannot be expressed in a configuration-based specification, and is therefore not captured in the above definition of a self-stabilizing protocol.

Another, more general, way to view the specification of a protocol is as describing the set of behaviours $\mathcal{B} \subseteq \mathcal{E}$ the protocol should abide: i.e. any execution of the protocol should belong to its specification. In this setting a protocol is *self-stabilizing* to specification $\mathcal{B}$ if for all executions E of the protocol there exists an $i \geq 0$ such that all executions of the protocol starting in configuration c_i belong to $\mathcal{B}$, i.e. if $(\forall E \in \mathcal{E}, \exists i \geq 0, \forall E_i = (c_i^E \ldots) \in \mathcal{E} : E_i \in \mathcal{B})$. In other words, a self-stabilizing protocol eventually cannot violate its specification.

Using this as a starting point, Burns et. al. [BGM93] call a protocol *pseudo-stabilizing* to specification $\mathcal{B}$ if for all executions $E = (c_j)_{j \geq 0}$ of the protocol there exists an $i \geq 0$ such that $(c_j^E)_{j \geq i} \in \mathcal{B}$, i.e. if $(\forall E \in \mathcal{E}, \exists i \geq 0 : (c_j^E)_{j \geq i} \in \mathcal{B})$. In other words, a pseudo-stabilizing protocol eventually will not violate its specification. Pseudo-stabilizing protocols are weaker than self-stabilizing protocols. The first *will not* violate its specification, whereas the other *cannot* violate its specification. As the second statement depends on the assumption that the system will not be disrupted by another transient error, the difference seems artificial in practice. As Burns et.al. [BGM93] observe that it is much easier to make pseudo-stabilizing protocols than to make self-stabilizing protocols, one might favour the first. However, pseudo-stabilizing protocols are not required to stabilize within an certain amount of time. In fact one easily sees that if one can prove an upper bound on the stabilization time of a pseudo-stabilizing protocol, then this protocol is also self-stabilizing[3]. Thus pseudo-stabilizing protocols that are not self-stabilizing actually do not guarantee that the system will be legitimate even once, if infinite bursts of transient errors are considered.

One can build self-stabilizing protocols from scratch, or one can try to combine previous efforts to obtain more generally applicable protocols. Dolev et. al. [DIM93] introduced *fair protocol combination* as a useful tool to construct a self-stabilizing protocol from two, simpler, self-stabilizing protocols. Informally speaking, fair protocol combination combines a master-protocol PM—that stabilizes to a certain specification provided certain external conditions

[3]Observe that for each execution E_i starting in c_i^E of execution E, there exists an execution $E' = (c_i')_{i \geq 0}$ such that $E_i = (c_j^{E'})_{j \geq i}$

hold—with a slave-protocol PS that stabilizes to executions in which exactly these conditions hold. The resulting protocol is a self-stabilizing version of PM without requiring those external conditions. In the combined protocol the steps of both protocols are taken alternately; the states of both protocols are merged so that the master protocol can read the state of the slave protocol. For more details we refer to Dolev et. al. [DIM93]. Their construction of a self-stabilizing mutual exclusion protocol for arbitrary graphs is a good example of the use of fair protocol combination. It combines a self-stabilizing mutual exclusion protocol that only operates on tree-shaped graphs (the master protocol) with a self-stabilizing spanning tree protocol for arbitrary graphs (the slave protocol). The slave protocol ensures that the master protocol is eventually run a tree-shaped graph, which in turn guarantees that the combined protocol will eventually satisfy the mutual exclusion requirements.

4. Ring-Orientation in the Link-Register Model

In the ring-orientation problem it is required that all nodes agree on an orientation. That is, all nodes should choose a left and right neighbour such that the left neighbour of each node considers that node to be its right neighbour. Thus a self-stabilizing ring-orientation protocol must stabilize to the set of executions

$$\begin{aligned}\mathcal{B}_{RO} = \{E = (c_0 c_1 \ldots) \mid \\ &(\forall q \in V_R, \exists p \in V_R : left(q) = p \wedge right(p) = q \text{ is stable in } c_0 \text{ of } E\}\end{aligned}$$

To obtain a uniform deterministic self-stabilizing ring-orientation protocol for odd-length rings, we use the randomized ring-orientation protocol of Israeli and Jalfon [IJ93]. This protocol is composed of two layers, combined using fair protocol combination, both operating in the link-register model under the distributed daemon. The lower layer is a (what we will call) randomized self-stabilizing *neighbour-ordering* protocol that will stabilize to a state in which any two neighbours agree on the order between them. The second layer is a deterministic self-stabilizing ring-orientation protocol assuming that all neighbours are mutually ordered, using only a constant number of states per node.

Intuitively the second layer of the Israeli and Jalfon protocol operates as follows. Initially certain nodes may hold a token, while others may be about to create tokens. If the ring is not oriented initially, at least one token is present or about to be created. However, once a node has created a token, it will never create a new token[4]. Each token has a fixed direction in which it travels around the ring. Whenever a node passes a token, this token directs the node. If two tokens meet, one of them will be eliminated based on the neighbour-ordering between the two nodes. Due to the elimination of opposite tokens, eventually all tokens will travel in the same direction around the ring, and thus the ring will eventually be oriented. For details we refer to [IJ93].

As an alternative for the lower layer we present a uniform deterministic self-stabilizing neighbour-ordering protocol for odd-length rings, using only a constant number of states per node, in the link-register model under the distributed daemon. Combining this with the second layer of the Israeli and Jalfon protocol, yields the desired self-stabilizing ring-orientation protocol for odd-length rings using a constant number of states per node.

[4]This may seem to conflict with self-stabilization, but is easily achieved if one makes sure that no configuration in which a token may be generated by a node q can be the result of a step of q.

4.1 Neighbour-Ordering in the Link-Register Model

In the neighbour-ordering problem it is required that eventually any two neighbours p and q agree on an order $<$ between them, and that once $p < q$ holds, $p < q$ remains to hold forever. That is, a self-stabilizing neighbour-ordering protocol must stabilize to the set of executions

$$\begin{aligned}\mathcal{B}_{NO} = \{&E = (c_0 c_1 \ldots) \in \mathcal{E} \mid \\ &(\forall pq \in E_R : (p < q \text{ stable in } c_0 \text{ of } E) \vee (p > q \text{ stable in } c_0 \text{ of } E))\}\end{aligned}$$

In this section we develop a uniform deterministic self-stabilizing neighbour-ordering protocol operating in the link-register model under the distributed daemon[5]. It is based on certain properties of odd-length rings, that we derive in the next few paragraphs.

Define for neighbours $p, q \in V_R$ the relations $\ll$ and $\equiv$ by

$$\begin{aligned} p &\ll q \quad \text{if } rank_p(q) < rank_q(p) \\ p &\equiv q \quad \text{if } rank_p(q) = rank_q(p) \end{aligned}$$

Label the edges $pq \in E_R$ (recall that E_R only contains clockwise edges) with $O \in \{\ll, \equiv, \gg\}$ such that $p \; O \; q$ if and only if pq is labelled O. If we consider chains $q_0 \ldots q_{k+1}$, then

$$q_0 \gg q_1 \equiv \cdots \equiv q_k \ll q_{k+1} \text{ implies } k \text{ is even} \tag{4.1}$$

$$q_0 \gg q_1 \equiv \cdots \equiv q_k \gg q_{k+1} \text{ implies } k \text{ is odd} \tag{4.2}$$

This leads us to the following claim.

Claim 4.1 *For any ring R with edges labelled as defined above, the number of edges labelled $\equiv$ is even.*

Proof: Let R be an arbitrary ring, and consider neighbouring nodes p, q, r and s (in that order). If $q \ll r$ it is easily seen that $rank_q(r) = rank_r(s)$ and $rank_r(q) = rank_q(p)$. Thus we can remove all edges not labelled $\equiv$—merging their endpoints—and the remaining ring has the same number of $\equiv$-labelled edges as R has. But obviously if all edges in the remaining ring are labelled $\equiv$, the ring must have an even number of edges. ■

From this claim it follows that any odd-length ring exhibits a certain asymmetry. The construction of our protocol will take this asymmetry as point of departure.

Corollary 4.2 *Let R be an odd-length ring, whose edges are labelled as defined above. Then there exists at least one edge labelled with $\ll$ or $\gg$, and the parity of the number of edges labelled $\gg$ is unequal to the parity of the number of edges labelled $\ll$.*

[5]Under the *central daemon* neighbour-ordering is easily achieved deterministically. Let each register R_{pq} contain a value in the set $\{0, 1\}$, and define $p < q$ if $R_{pq} < R_{qp}$. For any edge pq run the following protocol on p (and q, with p and q interchanged)

```
**if** $R_{pq} = R_{qp}$ **then** invert $R_{pq}$
```

This protocol will stabilize in exactly 1 round.

Now the straightforward approach to construct a self-stabilizing protocol for ring-orientation of odd-length rings might be one in which one generates tokens on $\ll$ or $\gg$-labelled edges, letting them travel in the direction of the $\ll$ or $\gg$. This will not work, however, because the parity-difference between $\ll$ and $\gg$-labelled edges on which it is based may be destroyed by two causes: initially extra tokens may be present on $\equiv$ labelled links, and each $\ll$-labelled edge has to generate infinitely many tokens, because it can never know it generated one. Apparently our previous corollary alone will not do: we are in need of an additional property of odd-length rings

Definition 4.3 *Mark an arbitrary, non zero, number of edges originally labelled $\equiv$ with $\bowtie$ instead. Then a clockwise chain $p_{l+1}p_l \ldots p_0q_0 \ldots q_rq_{r+1}$ is called a* $\bowtie$-delimited chain $C^{\bowtie}$ *if the edges $p_{l+1}p_l$, p_0q_0 and q_rq_{r+1} are the only edges marked $\bowtie$ in $C^{\bowtie}$.*

Note that this definition also captures the case in which only one edge is marked $\bowtie$, because chains are allowed to span the ring more than once. This definition is used in the following lemma to expose yet another source of asymmetry on odd-length rings.

Lemma 4.4 *Let R be an odd-length ring, whose edges are labelled as defined above. Mark an arbitrary number of edges with $\bowtie$ according to the previous definition. Then there exists at least one $\bowtie$-delimited chain $p_{l+1}p_l \ldots p_0q_0 \ldots q_rq_{r+1}$ such that the parity of the number of edges between p_l and p_0 labelled $\gg$ is unequal to the parity of the number of edges between q_0 and q_r labelled $\ll$.*

Proof: Let λ be the total number of edges labelled $\gg$ and let ρ be the total number of edges labelled $\ll$. By Corollary 4.2 we know that λ is odd (even) whereas ρ is even (odd). Given a ring R marked with $\bowtie$, consider the set $\{C_i^{\bowtie}\}$ of all $\bowtie$-delimited chains along R. For each $C_i^{\bowtie}$ let l_i be the number of $\gg$ labelled edges left of the middle $\bowtie$, and let r_i be the number of $\ll$ labelled edges right of the middle $\bowtie$.

Since we only marked edges with $\bowtie$ that were labelled with $\equiv$, and as all other edges occur exactly once left and exactly once right of the middle $\bowtie$ of some $C_j^{\bowtie}$, we see that $\sum l_i = \lambda$ and $\sum r_i = \rho$. Assume λ odd and ρ even (the other case leads to the same result). Then the number of odd l_i must be odd, whereas the number of odd r_i must be even. Therefore there must be a chain $C_i^{\bowtie}$ for which the parity of l_i does not equal the parity of r_i. ■

Now we are ready to give an informal description of the neighbour-ordering protocol. Each register R_{pq} holds a field $dir \in \{0,1\}$ such that $p < q$ if and only if $R_{pq}.dir < R_{qp}.dir$. If $p \neq q$, neither p nor q will try to order the pair (p,q). If $p = q$, node q can try to order (p,q) by inverting $R_{qp}.dir$, but under the distributed daemon both p and q might try to order (p,q) simultaneously. In that case the net result will be zero, and if the daemon always schedules p and q simultaneously (p,q) will never become ordered.

To avoid the above livelock schedule we propose the following. First we make sure that for any p and q with $p \gg q$, only q tries to order (p,q). By Corollary 4.2 at least one such (p,q) exists, which means that eventually one pair of nodes will be ordered. To order the remaining pairs (p,q) with $p \equiv q$, we only allow nodes q that are already ordered with respect to their other neighbour r (i.e. nodes q with $q \neq r$) to try to order (p,q).

Now there are two cases to consider

1. $o = p = q \neq r$: p will not try to order (p, q) so q must be allowed to do so.

2. $o \neq p = q \neq r$: the situation for p and q is symmetric and livelock might still occur.

To enable q to tell case 1 and 2 apart, R_{pq} will store whether $o = p$ in a field $ord \in \{\neq, =\}$. To break the symmetry in case 2 we apply Lemma 4.4: the parity of $\gg$-labelled edges left of p must be unequal to the parity of $\ll$-labelled edges right of q for at least one pair (p, q) in case 2. We let each node maintain the parity of $\gg$-labelled edges coming in from the other side in a field $parity \in \{0, 1\}$, and only allow q to try to order (p, q) if the parity it holds equals 1.

4.1.1 The protocol Let p and r be the neighbours of node q. In the neighbour-ordering protocol each register R_{qp} has the following fields

- $youare \in \{1, 2\}$, to encode the ordering $\ll$ between p and q. This field is always set such that $R_{qp}.youare = rank_q(p)$
- $dir \in \{0, 1\}$, to encode the desired node-ordering $<$ between p and q.
- $ord \in \{\neq, =\}$, to tell p whether $q = r$ or not.
- $parity \in \{0, 1\}$, holding the parity of the number of $\ll$ labelled edges coming in from the other side (i.e. through r).

Thus the protocol uses $2^4 \times 2^4$ states per node. Define for neighbouring nodes p and q

$$\begin{array}{lcll} p & \ll' & q & \text{if } R_{pq}.youare < R_{qp}.youare \\ p & \equiv' & q & \text{if } R_{pq}.youare = R_{qp}.youare \\ p & < & q & \text{if } R_{pq}.dir < R_{qp}.dir \\ p & = & q & \text{if } R_{pq}.dir = R_{qp}.dir \end{array}$$

In the neighbour-ordering protocol all nodes run the same program, consisting of two subroutines. The subroutine for a node q with neighbours p and r to update the register used to communicate with p (i.e. R_{qp}) is presented in Figure 2. Node q runs a similar subroutine to update R_{qr} (obtained by swapping p and r). The subroutine consists of a set of guarded commands, denoted by **if**-statements. Whenever node q is scheduled by the distributed daemon to take a step, all commands (both for updating R_{qp} and R_{qr}) whose guards are true are executed. At the start of each step, q reads R_{pq} and R_{rq} once, and q will write the new contents for R_{qp} and R_{qr} once at the end of each step.

4.1.2 Proof of correctness Throughout the proof, let $(P_i)_{i \geq 0}$ be an arbitrary fair schedule under the distributed daemon, let c_0 be an arbitrary initial configuration, and let E be the execution they induce.

```
(a)   $R_{qp}.youare \leftarrow rank_q(p)$
                  [force $\gg'$ to correspond to $\gg$]
(b)   if $q = r$ then $R_{qp}.parity \leftarrow 0$
                  [anchor the $parity$-chain]
(c)   if $q \ll' r \wedge q \neq r$ then $R_{qp}.parity \leftarrow (R_{rq}.parity + 1) \bmod 2$
                  [increase $parity$ for incoming $\ll'$ from other side]
(d)   if $q \not\ll' r \wedge q \neq r$ then $R_{qp}.parity \leftarrow R_{rq}.parity$
                  [pass on $parity$ to other side]
(e)   if $q = r$ then $R_{qp}.ord \leftarrow$ '=' else $R_{qp}.ord \leftarrow$ '$\neq$'
                  [pass ordering to other side]
(f)   if $p = q$
                  [only try to order not yet ordered pairs]
(f1)  then if $p \gg' q$ then invert $R_{qp}.dir$
                  [note that only the head of $\gg'$ directs the arc]
(f2)       else if $p \equiv' q \wedge q \neq r \wedge R_{pq}.ord =$ '=' then invert $R_{qp}.dir$
                  [case $\cdots o \approx p = q \neq r \cdots$]
(f3)       else if $p \equiv' q \wedge q \neq r \wedge R_{pq}.ord =$ '$\neq$' $\wedge R_{qp}.parity = 1$ then invert $R_{qp}.dir$
                  [case $\cdots o \not\approx p = q \neq r \cdots$]
```

Figure 2: Neighbour-Ordering subroutine for node q to adjust R_{qp}

Claim 4.5 *If $p < q$ holds in a configuration c of E, then $p < q$ is stable in c of E.*

Proof: The only steps of p or q that can change $R_{pq}.dir$ or $R_{qp}.dir$ (and therefore the ordering between p and q) are (f1), (f2) and (f3). But these three steps are guarded by the condition $p = q$ (see (f)), so the result follows. ■

Thus it remains to be shown that there exists an i such that for all $pq \in E_R$ we have $p \neq q$ in configuration c_i of execution E. We prove this by exhibiting an upper bound on the number of rounds of E after which a configuration satisfying this condition is guaranteed to be reached. This also provides us with an upper-bound on the stabilization time, measured in rounds, of the protocol. Recall that C_i is the configuration at the start of round i in execution E.

Lemma 4.6 *For all $pq \in E_R$, if $p \not\equiv q$ then $p \neq q$ in C_2.*

Proof: First observe that after round 0 each node has taken step (a) at least once, so we have $p \equiv q \Leftrightarrow p \equiv' q$, and similarly $p \gg q \Leftrightarrow p \gg' q$ and $p \ll q \Leftrightarrow p \ll' q$ for all $pq \in E_R$. Clearly these properties are stable in C_1. Take an arbitrary edge $pq \in E_R$ with $p \not\equiv q$. Assume $p \gg q$ (the case $p \ll q$ is handled similarly). Then $p \gg' q$ during round 1, which means that p cannot apply (f1), (f2) or (f3) on R_{pq} during round 1. If $p \neq q$ in C_1, then we are done according to Claim 4.5. Otherwise, q will apply (f1) on R_{qp} in round 1, setting $p \neq q$. ■

This leaves pairs (p, q) with $p \equiv q$. We first show that every node q faithfully conveys its order-relation with neighbour r to the other neighbour p.

Claim 4.7 *Let p and r be the neighbours of q. If $q \neq r$ is stable from C_i to C_j, then $R_{qp}.ord =$ '$\neq$' is stable from C_{i+1} to C_j. Similarly, if $q = r$ is stable from C_i to C_j, then $R_{qp}.ord =$ '$=$' is stable from C_{i+1} to C_j.*

Proof: If $q \neq r$ is stable from C_i to C_j, then q will apply step (e) in rounds i through $j-1$, setting $R_{qp}.ord =$ '$\neq$'. Thus $R_{qp}.ord =$ '$\neq$' is stable from C_{i+1} to C_j. The case for $q = r$ is handled similarly. ■

The next claim shows that if for a certain pair of nodes (r, p) we have $r = p$ for l rounds, then all nodes q with distance $k < l$ from p (without another equal pair of nodes in between) correctly store the parity of $\gg$ (i.e. pointing away from p) labelled edges between p and q.

Claim 4.8 *Suppose $r = p_0$ is stable from C_i to C_{i+l}, where $i \geq 2$ and $l \geq 0$. Pick $k < l$ such that on the chain $rp_0p_1 \ldots p_k$ along the ring we have $p_{i-1} \neq p_i$ for all i with $1 \leq i \leq k-1$. Define $\sigma_k = \left|\{p_i \mid 1 \leq i \leq k-1 \wedge p_{i-1} \gg p_i\}\right|$, i.e. the number of $\gg$ labelled edges between p_0 and p_{k-1}. Then $R_{p_{k-1}p_k}.parity = \sigma_k \bmod 2$ is stable from C_{i+k} upto C_{i+l}.*

Proof: Proof by induction on k. In the base case $k = 1$ we have $\sigma_1 = 0$. Since $r = p_0$ holds during round i through $i+l-1$, p_0 applies step (b) in these rounds, setting $R_{p_0p_1}.parity = 0$ as required. Thus this property is stable up to C_{i+l}.

Assume the induction hypothesis holds for $k < l-1$ and assume $p_{k-1} \neq p_k$. Then $R_{p_{k-1}p_k}.parity = \sigma_k \bmod 2$ from C_{i+k} up to C_{i+l}. Then there are two cases to consider. If $p_{k-1} \gg p_k$, then $p_{k-1} \gg' p_k$ stable in C_{i+k} (as $i \geq 2$) and $\sigma_{k+1} = \sigma_k + 1$. So p_k takes step (c) in rounds $i+k$ through $i+l-1$, setting $R_{p_kp_{k+1}}.parity$ to $(R_{p_{k-1}p_k}.parity + 1) \bmod 2$ in these rounds as required. If $p_{k-1} \not\gg p_k$, then $p_{k-1} \not\gg' p_k$ stable in C_{i+k} (as $i \geq 2$) and $\sigma_{k+1} = \sigma_k$. So p_k takes step (d) in rounds $i+k$ through $i+l-1$, setting $R_{p_kp_{k+1}}.parity$ to $R_{p_{k-1}p_k}.parity$ in these rounds as required. ■

Theorem 4.9 *The neighbour ordering protocol stabilizes on an odd-length ring, under the distributed daemon, to a configuration in which for any two neighbours p, q, $p \neq q$. Furthermore, once $p < q$, then $p < q$ holds forever. The system stabilizes in $O(n^2)$ rounds.*

Proof: By Lemma 4.6 in configuration C_2 we have for all edges $pq \in E_R$ with $p \not\equiv q$ that $p \neq q$. By Corollary 4.2 there exists at least one such edge.

Suppose that in configuration C_i there exist x triplets o, p, q with $o = p = q$. By the above observation for at least one we actually have the quartet $o = p = q \neq r$. Suppose $o = p$ is stable from C_i to C_{i+2}. Then by Claim 4.7, $R_{pq}.ord =$ '$=$' and $R_{qp}.ord =$ '$\neq$' are stable from C_{i+1} to C_{i+2}. Then in round $i+2$, p cannot apply any of the steps (f1), (f2) and (f3). If in C_{i+1} $p = q$ holds, q will apply (f2) in round $i+2$, setting $p \neq q$. If in C_{i+1} $p = q$ does not hold, then by Claim 4.5 in C_{i+2} we have $p \neq q$. If $o = p$ is not stable from C_i to C_{i+2}, then $o \neq p$ in C_{i+2}. In either case, the number of triplets $o = p = q$ in C_{i+2} will be at most $x - 1$. As the number of triplets is at most $n - 3$ (n is the number of nodes), in configuration C_{n-1} there will only be edges pq with $p = q$ such that $o \neq p = q \neq r$.

Suppose that in configuration C_m, $m \geq n$, there are y edges pq with $p = q$ (and hence $o \neq p = q \neq r$). Now mark all edges $pq \in E_R$ with $p = q$ (and hence $p \equiv q$) with $\bowtie$. By

Lemma 4.4 there exists at least one $\bowtie$-delimited chain $p_{l+1} = p_l \cdots p_0 = q_0 \cdots q_r = q_{r+1}$ such that the parity of the number of edges between p_l and p_0 labelled $\gg$ is unequal to the parity of the number of edges between q_0 and q_r labelled $\ll$. Note that for all edges pq between p_l and p_0 or between q_0 and q_r we have $p \neq q$.

Let $\mu = 1 + \max(l, r)$. Consider the above chain and suppose $p_{l+1} = p_l$ and $q_r = q_{r+1}$ are stable from C_m to $C_{m+\mu+1}$. Then by Claim 4.8, $R_{p_0 q_0}.parity \neq R_{q_0 p_0}.parity$ in round $m + \mu$. By Claim 4.7 also $R_{p_0 q_0}.ord =$ '$\neq$' and $R_{q_0 p_0}.ord =$ '$\neq$' in round $m + \mu$. If $p_0 = q_0$ in $C_{m+\mu}$, either p_0 or q_0 will apply step (f3) in round $m + \mu$ setting $p \neq q$ in $C_{m+\mu+1}$. If $p_0 \neq q_0$ in $C_{m+\mu}$, then by Claim 4.5 $p_0 \neq q_0$ in $C_{m+\mu+1}$. In either case, the number of edges pq with $p = q$ in $C_{m+\mu+1}$ is at most $y - 1$. Also, if $p_{l+1} = p_l$ or $q_r = q_{r+1}$ is not stable from C_m to $C_{m+\mu+1}$, then the number of pairs with $p = q$ in $C_{m+\mu+1}$ is at most $y - 1$.

As $\mu + 1 \leq n$, and $y \leq \lfloor \frac{n}{2} \rfloor$ in C_n, after round $n + n * \lfloor \frac{n}{2} \rfloor$ the system will certainly be in a legitimate configuration. ■

5. Ring-Orientation in the State-Reading Model

In this section we present a uniform deterministic self-stabilizing ring-orientation protocol for odd-length rings operating in the state-reading model under the central daemon, using only a constant number of states per node. This is the best we can hope for, as Israeli and Jalfon [IJ93] have already shown that such protocol is impossible under the distributed daemon, and under the central daemon if the ring-length is even. Most ring-orientation protocols operate by forwarding a token with a fixed direction around the ring. But in the state-reading model both neighbours of a node read the same state. If that node were to hold a token with a direction, it is not immediately obvious how to forward that token to the one neighbour it points to without possibly forwarding it to the other neighbour as well. In view of this observation it is perhaps surprising that it is possible at all to orient a ring in the state reading model.

Intuitively the protocol operates as follows. Let each node have a *colour* taken from the set $\{0, 1\}$. Try to give neighbouring nodes alternating colours by inverting the colour of a node if it has the same colour as both its neighbours. Of course on an odd-length ring this is never completely possible, so we are bound to end up with patterns like 001 and 110 around the ring[6]. Call such patterns tokens. The idea is to make these tokens travel around the ring, orienting the nodes they visit. This requires us to impose a direction on tokens, for which we let each node store a *phase* taken from the set $\{+, -\}$. A token is directed if the nodes with equal colour have opposite phase: then the direction of the token points from the node with phase $+$ to the node with phase $-$. Undirected tokens can be directed by letting the middle node in a token invert its phase.

Let us write 0_- for a node with colour 0 and phase $-$, and consider the pattern $0_+ 0_- 1_- 0_-$ around the ring. If we allow the second node to change its state to 1_+ we get the pattern $0_- 1_+ 1_- 0_-$: the token has moved one step in its direction. Now let a node also maintain its orientation, taken from $\{\leftarrow, \rightarrow\}$, for instance by storing the rank of the neighbour the

[6]This trick cannot be applied immediately under the distributed daemon, because the distributed daemon is free to schedule all nodes simultaneously. If all nodes in the ring have the same colour, then under such a schedule all nodes will simultaneously invert their colour.

δ	p	q	r	q'
(a)	0	0	0	1_-
(b)	0	1	0	1_-
(c)	1	1	1	0_-
(d)	1	0	1	0_-

δ	p	q	r	q'
(e)	0_+	0_-	1_-	$\overrightarrow{1_+}$
(f)	1_+	1_-	0_-	$\overrightarrow{0_+}$

δ	p	q	r	q'
(g)	0_-	0_-	1	0_+
(h)	0_+	0_+	1	0_-
(i)	1_-	1_-	0	1_+
(j)	1_+	1_+	0	1_-

Figure 3: Ring-Orientation program for node q

head of the arrow points to. If a token moves one step, the node changing colour is oriented into the direction of the token. Our protocol depends on the fact that a token always keeps the same direction, until it is eliminated. Then in the situation $0_+0_-1_+0_-$ it is unwise to allow the second node to change its state to 1_+ as this would yield the situation $0_+1_+1_+0_-$. Here the token becomes undirected, and thus may decide to invert its direction: $0_+1_-1_+0_-$. Therefore, in situations like $0_+0_-1_+0_-$ we must wait until the third node sets its phase to $-$.

We have already seen that in almost all configurations (except for the special case in which all nodes have the same colour) at least one token exists. If we can make sure that eventually all tokens travel in the same direction around the ring, the ring will eventually become oriented. Now consider what happens if two tokens with opposite direction meet. Then we will see the a sequence of patterns similar to

$$0_+0_-1_-0_-0_+ \longrightarrow 0_+1_+1_-0_-0_+ \longrightarrow 0_+1_+0_+0_-0_+$$

Other nodes in this example may take a step instead, but this leads to essentially the same situation. As we already try to colour the nodes alternatingly, the middle node of three nodes with equal colour will eventually invert its colour, thus eliminating both tokens.

5.1 The protocol

Let p and r be the neighbours of q. Each node q stores its whole state (i.e. its colour, phase and orientation) in register R_q. In the ring-orientation protocol, all nodes q run the same program, presented in Figure 3 as a state-transition function δ. The tables list the applicable steps for a node q with neighbours p and r, depending on the state of p, q and r (i.e. the contents of R_p, R_q and R_r). The entries under q' list the new state of q after applying δ. To reduce the size, symmetric steps (with p and r interchanged) are not shown. So steps (e) through (j) actually represented two steps. If only some components of a state are specified, the other components may have an arbitrary value. If in the new state the direction is not specified, it should equal the direction of q in the old state. In cases (e) and (f), the direction of q in the new state should be read as pointing from p to r. Node q can always determine this direction because in these steps p and r hold opposite colours. The rank of p and r is used to encode the direction.

Each table contains related steps. The first table lists the steps that try to colour the ring alternately, and to make sure that alternately coloured nodes have phase $-$. The second table lists the steps responsible for forwarding the token, and the third table lists the steps that break the symmetry in possible tokens by imposing one direction on them.

5.2 Proof of correctness

Throughout the proof, let $(P_i)_{i \geq 0}$ be an arbitrary fair schedule under the central daemon, let c_0 be an arbitrary initial configuration, and let E be the execution of the protocol they induce. If $n = 1$, the protocol is trivially correct, so in the remainder of the proof we assume $n \geq 3$. We prove correctness of the algorithm in stages. In each stage we define a set of configurations and show that the protocol will converge to this set, when started in a configuration from the set of the previous stage. All sets are shown to be closed under transitions of the protocol. In the final stage the set of configurations will contain exactly those that are oriented. To describe the configurations in a set we use regular expressions.

Definition 5.1 *A configuration* matches *a regular expression L if the concatenation of the states of some chain (clockwise or anticlockwise) of length n matches the regular expression. A regular expression L is* closed *under steps of the protocol if for all configurations c that match L, c' with $c \rightarrow_P c'$ for arbitrary P matches L as well.*

Define the regular expression L_1 by

$$\begin{aligned} L_1 &= (O_1 I_1)^* \\ O_1 &= 0 \mid 0_+0_-0_+ \mid 0_-0_+ \mid 0_+0_- \\ I_1 &= 1 \mid 1_+1_-1_+ \mid 1_-1_+ \mid 1_+1_- \end{aligned}$$

Lemma 5.2 *Starting in an arbitrary configuration, we eventually reach a configuration matching L_1. Furthermore, L_1 is closed.*

Proof: First we prove closure. Let a node q with colour 0 take a step (the case of a node with colour 1 taking a step is handled similarly). Then q belongs to a chain matching O_1, because the current configuration matches L_1, and left and right of this chain there are chains matching I_1. We must show that any step of q in any configuration matching L_1 yields a configuration matching L_1. That is, starting with a chain matching $I_1O_1I_1$ we get a chain matching $I_1O_1I_1$ or similar. This leads to the following case analysis

101	Becomes 10_-1 by step (d), which matches $I_1O_1I_1$.
$10_+0_-0_+1$	Only the middle node, with state 0_-, can move according to step (a), yielding $10_+1_-0_+1$ matching $I_1O_1I_1O_1I_1$.
10_+0_-1	Only node 0_- can move, but only if the right 1 has phase $-$. But for 1_- to match I_1 we must actually have one of the the following: $10_+0_-1_-0$ which becomes $10_+1_+1_-0$ after applying step (e), or $10_+0_-1_-1_+0$ which becomes $10_+1_+1_-1_+0$ after applying step (e). Both chains match $I_1O_1I_1O_1$.
10_-0_+1	See previous case

For progress, consider C_1, the configuration at the end of round 0 of E. Now suppose C_1 does not match L_1. Then C_1 contains one of the following patterns. Since all nodes must have taken at least one step in round 0, we derive a contradiction from each of the patterns. In the following let $\boxed{0_-}$ denote the node taking the last step in round 0 of all nodes shown, and consider only patterns consisting of 0's. Considering the patterns consisting of 1's leads to the same analysis.

$\boxed{0_-}0_-$	Only (c), (d) or (h) lead to state 0_-, but none of them are applicable if one of the neighbours has state 0_-.
$\boxed{0_+}0_+$	Only (f) or (g) lead to state 0_+, but none of them are applicable if one of the neighbours has state 0_+.
$\boxed{0_-}0_+0_-$	Only (c), (d) or (h) lead to state 0_-. But (c) and (d) are not applicable if one of the neighbours has state 0_+, and if (h) were applicable, then the pattern just before this was $0_+0_+0_-$. Either one of the nodes with state 0_+ took the last step but one, which leads to the second case above.
$0_-\boxed{0_+}0_-$	Only (f) or (g) lead to state 0_+, but none of them are applicable if both neighbours have state 0_-.

A similar analysis also shows that none of the indicated last steps could have been void steps. This proves that C_1 matches L_1. ■

All configurations matching the regular expression L_2 defined below only contain tokens in one direction.

$$L_2 = (O_2 I_2)^* \qquad O_2 = 0 \mid 0_+0_- \qquad I_2 = 1 \mid 1_+1_-$$

Lemma 5.3 *From a configuration matching L_1, we eventually reach a configuration matching L_2. Furthermore, L_2 is closed.*

Proof: Closure of L_2 is proven as in Lemma 5.2, noting that the third case changes to

10_+0_-1	Only node 0_- can move, but only if the right 1 has phase $-$. But for 1_- to match I_2 we must actually have the following: $10_+0_-1_-0$ which becomes $10_+1_+1_-0$ after applying step (e). This string matches $I_2O_2I_2O_2$.

To prove that the system eventually reaches a configuration matching L_2, consider a configuration matching L_1. This configuration may fail to match L_1 for two reasons. First of all it may contain patterns like $0_+0_-0_+$ and $1_+1_-1_+$, and secondly it may contain tokens (0_+0_- and 1_+1_-) in opposite directions.

Now consider two tokens pointing towards each other, with all nodes in between having alternating colours. If some node in between takes a step, it will not change its colour, but will set its phase to $-$. After one round all these nodes will have phase $-$. In the next round, the tokens will move towards each other (because both have to apply step (e) or (f)). So the distance between them eventually becomes 0, i.e. a configuration like $0_+0_-1_-1_+$. If one of the middle nodes takes a step, this yields $0_+0_-0_+$ or $1_+1_-1_+$. This proves that eventually all opposing tokens are eliminated.

So we eventually reach a configuration in which all tokens have the same direction, and in which patterns $0_+0_-0_+$ and $1_+1_-1_+$ may occur. As these chains can only be created by tokens with opposite directions, no further chains of this kind will be created. These chains will eventually disappear, as the middle node can (and will) apply steps (a) or (c). Once all these chains have disappeared, we have reached a configuration matching L_2. ■

All configurations matching the regular expression L_3 defined below are oriented.

$$L_3 = (O_3 I_3)^* \qquad O_3 = \overrightarrow{0} \mid \overrightarrow{0_+} \; \overrightarrow{0_-} \qquad I_3 = \overrightarrow{1} \mid \overrightarrow{1_+} \; \overrightarrow{1_-}$$

Lemma 5.4 *From a configuration matching L_2, we eventually reach a configuration matching L_3. Furthermore, L_3 is closed.*

Proof: Closure is proven as in Lemma 5.3, noting that the orientation does not change in either case. To prove that the system eventually reaches a configuration matching L_3, note that because the ring has odd length, at least one token must exist, and that for at least one token we have 0_+0_-10 or 1_+1_-01. Thus the head of at least one token can eventually take a step, moving the token one position into the direction of the token, and directing the former head as well. Eventually one token must have travelled around the ring completely, at which time the ring is oriented. ■

From Lemma 5.2, 5.3 and 5.4 we easily obtain the following theorem.

Theorem 5.5 *The ring-orientation protocol stabilizes, under the central daemon, to a configuration in which the ring is oriented, provided the length of the ring is odd.*

6. On the Equivalence of Self-Stabilizing System Models

An overwhelming amount of models for distributed systems can be found in the literature, some of which only differ on seemingly minor points. This diversity is caused by the fact that slight alterations to a model may have a huge effect on the (im)possibility or (in)efficiency of certain protocols[7]. Also, the initial inability to obtain results in a certain model may have led people to adjust assumptions somewhat. And last but not least, there is no general consensus on what constitutes a real distributed system let alone how it should be modelled formally.

A major challenge is to explore these different models, and find conditions or application areas under which these models are equivalent. In the area of self-stabilization, research in this area has already been started by Gouda et. al. [GHR90]. In this section we will show that on oriented rings the link-register and state-reading model are equivalent. Using the results of the previous sections, we also show that as a consequence, the link-register model and the state-reading are eventually equivalent on odd-length rings under the central daemon.

What do we mean by equivalence between models? Several definitions seem to be appropriate (cf. [GHR90, LV93]). Intuitively two models are equivalent if they are of equal strength: whatever is possible in one model, is also possible in the other, and vice versa. We adopt the following formal definitions.

Definition 6.1 *Protocol P_1* simulates *protocol P_2 if there exists a mapping μ such that for all executions E_1 of P_1, $\mu(E_1)$ is an execution of P_2. Protocol P_1* eventually simulates *protocol P_2 if there exists a mapping μ such that for all executions E_1 of P_1, a suffix of $\mu(E_1)$ is an execution of P_2. Protocols P_1 and P_2 are* (eventually) equivalent *if both (eventually) simulate each other. System model M_1* (eventually) simulates *system model M_2 if for all protocols P_2 on M_2 there exists a protocol P_1 on M_1 that (eventually) simulates P_2. System models M_1 and M_2 are* (eventually) equivalent *if both (eventually) simulate each other.*

[7] As exemplified by the large body of literature on consensus.

We choose to show simulation of M_1 by M_2 by giving a general method to convert a protocol in M_1 to a simulating protocol in M_2 and describing the mapping μ from executions in M_2 to M_1. As in the definition, both models are equivalent if we can show that both simulate each other.

Theorem 6.2 *For oriented rings, the state-reading model and the link-register model are equivalent*

Proof: A protocol in the state-reading model is easily transformed into an equivalent protocol in the link-register model, by changing all writes to R_q into writes of the same value to R_{qp} and R_{qr}, and changing all reads from R_p to reads from R_{pq}. Initially the contents of R_{qp} and R_{qr} should equal the contents of R_q. Then μ maps all executions in the link-register model to executions in the state-reading model by mapping the contents of R_{qp} and R_{qr} (that by construction always hold the same value) to the contents of R_q.

A protocol in the link-register model is transformed into an equivalent protocol in the state-reading model as follows. Split in the state-reading model the register R_q into two fields *left* and *right*, each containing the same values as R_{qp} and R_{qr} in the link-register model. Now let p and r be the left and right neighbours of q, respectively. Then change the writes to R_{qp} to writes to $R_q.\mathit{left}$ and the writes to R_{qr} to writes to $R_q.\mathit{right}$. Change reads from R_{pq} to reads from $R_p.\mathit{right}$ and reads from R_{rq} to reads from $R_r.\mathit{left}$. Then μ maps all executions in the state-reading model to executions in the link-register model by mapping the contents of $R_q.\mathit{left}$ to the contents of R_{qp} and the contents of $R_q.\mathit{right}$ to the contents of R_{qr}. ■

From the self-stabilizing ring-orientation protocols presented in the previous section, we easily obtain the following corollary.

Corollary 6.3 *For odd-length rings, the link-register and state-reading model are eventually equivalent under the central daemon.*

Simply combine the simulation in the proof of Theorem 6.2 with the ring-orientation protocol for the state-reading model. Note that the simulation of the state-reading protocol by the link-register protocol ensures that after one step of q after an error, the contents of R_{qp} and R_{qr} are equal. The ring-orientation protocol requires a central daemon.

Observe that a similar corollary cannot be obtained for odd-length rings under the distributed daemon, because Israeli and Jalfon already showed that no ring in the state-reading model can be oriented under the distributed daemon.

7. Conclusions and Further Research

We have shown that uniform deterministic self-stabilizing ring-orientation protocols using only a constant number of states per node exist for odd-length rings, both in the link-register model under the distributed daemon, and in the state-reading model under the central daemon. Further research might be directed at deriving similar protocols to other graphs with a regular structure, like cliques or hypercubes. We have also shown that the link-register model and the state-reading model are eventually equivalent on odd-length rings under the central daemon. We are unaware of similar theorems exploring the relation between the central daemon and the distributed daemon.

8. Acknowledgements
The author wishes to thank Marina Papatriantafilou and Philippas Tsigas for the discussions on this problem while they were visiting the CWI in Amsterdam.

References
[ASW88] Attiya, H., Snir, M., and Warmuth, M. K. Computing on an anonymous ring. *Journal of the ACM* **35**, 4 (1988), 845–875.

[BGM93] Burns, J. E., Gouda, M. G., and Miller, R. E. Stabilization and pseudo-stabilization. *Distributed Computing* **7**, 1 (1993), 35–42.

[BP89] Burns, J. E., and Pachl, J. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems* **11**, 2 (1989), 330–344.

[Bur87] Burns, J. E. Self-stabilizing rings without demons. Tech. Rep. GIT-ICS-87/36, School of Information and Computer Science, Georgia Institute of Technology, 1987.

[Dij74] Dijkstra, E. W. Self-stabilizing systems in spite of distributed control. *Communications of the ACM* **17**, 11 (1974), 643–644.

[Dij82] Dijkstra, E. W. Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, 1982, pp. 41–46.

[DIM93] Dolev, S., Israeli, A., and Moran, S. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing* **7**, 1 (1993), 3–16.

[GHR90] Gouda, M. G., Howell, R. R., and Rosier, L. E. The instability of self-stabilization. *Acta Informatica* **27**, 8 (1990), 697–724.

[IJ93] Israeli, A., and Jalfon, M. Uniform self-stabilizing ring orientation. *Information and Computation* **104**, 2 (1993), 175–196.

[LV93] Lynch, N. A., and Vaandrager, F. W. Forward and backward simulations. Part I: Untimed systems. Tech. Rep. CS-R9313, Center for Mathematics and Computer Science (CWI), Amsterdam, 1993.

[San84] Santoro, N. Sense of direction, topological awareness and communication complexity. *ACM SIGACT News* **16**, 2 (1984), 50–56.

[SP87] Syrotiuk, V., and Pachl, J. A distributed ring orientation algorithm. In *Proc. of the 2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), pp. 332–336.