# The uniform self-stabilizing orientation of unicyclic networks

H. James Hoover[*]
Piotr Rudnicki[†]

Department of Computing Science
University of Alberta

Technical Report TR 91–02

1991 June 15
Revised 1991 August 26

**Abstract**

We present a very simple protocol for the self-stabilizing orientation of a unicyclic network of uniform processors. It has the same $O(n^2)$ performance as the Israeli and Jalfon protocol for rings but is much simpler to state and furthermore operates under the weaker model of read/write demon asynchronicity. We also elucidate some of the techniques used in the design of such protocols, but not often stated in the literature.

In addition, we propose a cleaner characterization of the various models used for such protocols by separating the issues of operation atomicity from the behaviour of the scheduling adversary. This eliminates the need to assume either a fair or proper scheduler when reasoning about the protocol.

[*]Research supported by the Natural Sciences and Engineering Research Council of Canada grant OGP 38937. Author's address: Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, e-mail address: `hoover@cs.ualberta.ca`

[†]Research supported by the Natural Sciences and Engineering Research Council of Canada grant OGP 09207 Author's address: Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, e-mail address: `piotr@cs.ualberta.ca`

## 1 Introduction

In [IJ91], Israeli and Jalfon consider the problem of the uniform self-stabilizing orientation of a ring of processors so that the processors achieve consistent notions of left and right. Each processor is a randomized finite state machine which can communicate directly only with its two neighbours. All processors are identical and anonymous (i.e. they lack any distinguishing feature such as a processor id), and they may be started in arbitrary states. Furthermore, the processors run asynchronously and an atomic transition consists of reading all input ports, changing the state, and then writing all output ports.

In their paper, they show that no deterministic protocol exists for the self-stabilizing orientation of a ring, and also give a randomized $O(n^2)$ expected time orientation protocol under the distributed demon. Their protocol is moderately complex. It is composed of two self-stabilizing sub-protocols, one of which is randomized, the other is deterministic. The sub-protocols are then combined into the main protocol using the Dolev, Israeli and Moran [DIJ90] technique for the fair combination of self-stabilizing protocols.

The notion of orienting a ring can be generalized to orienting any unicyclic graph (a ring with attached trees). The contribution of this paper is to give a simple protocol, both in intuition and implementation, for the self-stabilizing orientation of such a network under the read/write scheduling adversary. Our protocol has the same $O(n^2)$ performance as the Israeli-Jalfon protocol, but it does not require any sub-protocols, and has the property that it deadlocks exactly when it has stabilized.

## 2 Intuition

Consider the typical conference banquet. You are sitting at a round table containing $n$ diners. Immediately in front of you is the plate for your main course. To both sides of you, between you and your neighbours, are salads. Which salad do you choose? How do you ensure that everyone gets a salad?

In general your fellow diners will have conflicting views on which salad to take. The ensuing confusion, possibly involving some fights over the same salad, results in some diners getting salads, others going hungry, and unclaimed salads remaining on the table.

All of this unseemly behaviour could have been avoided had you all agreed beforehand upon a rule, say, that one's salad is always obtained from one's right. This of course assumes that each diner knows their left from right. If they do not, then obtaining a salad first requires that all diners consistently orient their handedness. Thus an algorithm for consistently orienting a ring of identical processors has immediate practical application to the problem of banquet dining.

One can distribute salads even when diners do not know left from right, and Figure 2.1 is a protocol that does so. This protocol assumes only two things: that diners can only communicate with their immediate neighbours; and that if two diners attempt to simultaneously take the same salad, exactly one wins. It is an interesting side effect of this protocol that in addition to their salad, all diners obtain a consistent notion of left and right. We will exploit this side effect to obtain our protocol.

```
while ( don't have a salad ) {
        Attempt to take a salad from your right.
        Reverse your notion of left and right.
}
while ( 1 ) {
        Wait for a salad to appear on your right.
        Pass the salad to your left, waiting until it is taken.
        Reverse your notion of left and right.
}
```

Figure 2.1: Informal salad protocol.

Let us call a diner without a salad *hungry*, and one with a salad *satisfied*. We say that two diners $A$ and $B$ have a *consistent orientation* if $B$ is to $A$'s left and $A$ is to $B$'s right, or vice versa. The salad protocol has a number of properties which we state without proof.

1. Hungry diners and unclaimed salads, if any, alternate around the table with satisfied diners between them.

2. Any interval containing only satisfied diners has a consistent orientation.

3. Any unclaimed salad moves around the table in one direction until it is claimed by a hungry diner. It cannot reverse direction.

4. When all salads are claimed, every diner is waiting for a salad to be passed to it from its right.

5. At most $O(n^2)$ salad passing operations are required before all diners are satisfied, and $\Omega(n^2)$ may be necessary depending on how simultaneous attempts at taking a salad are arbitrated.

Moving salads pass orientation information about the ring. The effect of passing a salad is to send a reorientation message to your neighbour, in effect saying that I had to change my orientation, perhaps you might also have to. Salads are removed from circulation by hungry diners exactly when they have served their purpose of consistently orienting a segment of the ring. This provides the intuition that forms the basis of our simple protocol, which we can informally state as follows:

> Look in the direction (to the right) that you *do not* expect a reorientation message to arrive. If one does arrive, then revise your notion of left and right, and pass on this reorientation information to your other neighbour.

## 3 Processor Networks

The idea of a consistent left-right notion for processors on a ring can be generalized to networks. But before doing so it is worth examining the various computational settings in which such protocols are studied.

The orientation protocols will be executed on a *processor network* in which processors are placed at the vertices of the network, and all communication between processors occurs over channels associated with the edges. Each processor at a vertex of degree $k$ in the network has exactly $k$ ports, numbered $0, 1, \ldots, k-1$. Each port consists of an input/output pair, which is connected to an input/output pair of a neighbour such that write conflicts are excluded.

A processor network is described by a graph $G = (V, E)$ with vertex set $V = \{0, \ldots, n-1\}$ and edge set $E \subset (V \times \{0, 1, 2, \ldots\}) \times (V \times \{0, 1, 2, \ldots\})$. Processor $P_v$ is associated with vertex $v$. Each port of $P_v$ is assigned to a distinct edge of $E$ incident on vertex $v$. An edge $((v, i), (w, j))$ of $E$ indicates that processor $P_v$ has its $i$ port connected, via a channel, to port $j$ of processor $P_w$. $G$ is required to be connected.

Processor networks come in many flavours, depending on how one answers the following questions:

- How powerful are the processors? Are they simple finite-state machines? Are they event driven? Are they randomized? Are the processors reliable? What kinds of failures do we consider?

- Are the processors homogeneous? Do they have unique processor ids? Do processors have any knowledge of the network topology, such as its size, maximum vertex degree, longest path, and so on?

- Do the processors in the network operate synchroneously? What kind of asynchronous behaviour is permitted? Do the channels have finite capacity? What kinds of messages are permitted on the channel? Are they blocking? Are they reliable? What kinds of errors occur?

- Is there a well-defined network state at the beginning of the protocol execution? Are individual processors required to recognize when the network is in a specific state? Does the network deadlock when it is in the required state? Is the protocol independent of network topology?

Regardless of flavour, we assume that the processors and channels are discrete-state devices, and thus the complete state description of a network is composed of the complete states of all the processors and channels.

As a network executes a protocol, it changes state, and as a result one obtains a notion of progress by simply observing the network state changes. However this simple notion of progress is essentially meaningless since processor networks at this lowest level of detail must, at the least, continually sample their inputs, and thus can change their states without making any progress towards the protocol goal. Thus any useful notion of execution progress must be based on a projection of certain components of the complete state of the network.

With this in mind, we informally define a *configuration* to be any projection of the state of a processor network. A configuration defines the observables of the network, and a processor not changing its observables is not doing anything under this configuration. All discussions of execution behaviour are done at the configuration level, and such discussions are incomplete without specifying the projection used to obtain the configuration. It is quite possible for a protocol to be correct under one notion of configuration, and incorrect under another, as protocol specifications are also expressed in terms of a configuration.

What are the basic execution models for a processor network? Each such model will dictate the nature of the interactions among concurrently operating processors, and consequently will determine the conceptual complexity of the protocol on that model. These interactions are affected by the atomic operations of the processor network and the way in which these operations are scheduled within the network.

Our basic model of execution uses polling, rather than being event driven. That is, processors are constantly sampling their environment. Otherwise the network could fault into a state in which all processors are waiting for events which cannot occur. Self-stabilization, that is the ability to withstand transient faults, is achievable only under polling.

A computation model is said to be *transient faulting* if during computation processors can fault to an arbitrary configuration. A correct protocol under a transient faulting computation model is said to be *self-stabilizing*.

Polling can be implemented implicitly or explicitly. One way of treating polling is to model processors as having non-total state transition functions. The state transition function maps the current state and states of the input ports to the next state. When the state transition function is defined the processor is *enabled*, otherwise the processor is *disabled*. Thus a disabled processor polls its inputs until becoming enabled at which time it continues to execute.

If one is interested in the finer details of reading and writing to ports, then implicit polling is not sufficient. Thus we require explicit polling by processors. In such a case, the state transition function of a processor is total, and there is no notion of a processor being enabled or disabled.

In addition, we require the ability to make random choices. If processors can make random state transitions, we say that the computation model is *randomized*, otherwise it is *deterministic*.

## 3.1 Operational Atomicity

Each individual processor or channel can perform certain operations which are atomic at the configuration level. All atomic operations take the device instantaneously from its current configuration to its next configuration. Since atomic operations are indivisible, two atomic operations cannot be temporally overlapped in the sense that one operation starts before the another completes. But two or more atomic operations can occur simultaneously at the same instant. In this way, the notion of an atomic operation for a single processor is extended to the network. An atomic network operation is a configuration-level notion involving the scheduler.

Processor networks can have the following kinds of operational atomicity:

*Single-Cycle Atomic:* The simplest approach to execution is to let each processor be a finite state machine, and let the channels be direct connections between ports. Each processor repeatedly executes a single atomic operation consisting of reading all of the input ports and then changing state on the basis of their value and the current state of the processor. The values that appear on the output ports are not explicitly written but instead are functions of the current state of the processor. Thus whenever it chages state so (potentially) do the outputs. In effect, your neighbour has direct access to some projection of your state.

*Read/Write-Subset Atomic:* A more general execution model distinguishes between reading an input and writing an output. In this setting the values appearing at an input port do not come directly from an output port, but are instead buffered by a register in the communication channel. There are two atomic port operations: read a (possible empty) subset of the input ports; and write a (possible empty) subset of the output ports. Since the read and write are atomic, it is impossible for a simultaneous write to a port to interfere with a read from the attached channel — the read will get the value prior to the write.

*Read/Write-Single Atomic:* This is like Read/Write-Subset Atomic except that only a single port can be read or written at a time.

*Read/Write Non-Atomic:* The most complex model attempts to capture the realistic situation in which reads and writes can be temporally overlapped and thus interfere with each other. For example a write occurring during a read could cause the read to return garbage. Thus there are four kinds of atomic port operations: start the read of a single port; complete the read of the port; start the write of a single port; complete the write of the port.

A totally different class of model occurs when the processor states are continuous functions of time. In their most general form, such continuous transition models must be described using differential equations. However, special cases of such models can be understood using the ideas of Lamport [Lam86a, Lam86b].

## 3.2 Scheduling Disciplines

The possible interactions among processors depend on their atomic operations and the possible ways in which simultaneous processor activity can occur. The possible concurrent atomic events are determined by the *scheduling discipline.* Processors have no control over the way in which their atomic operations are scheduled, and thus a correct protocol must be able to cope with any permissible schedule within the discipline. We distinguish the following kinds of scheduling disciplines:

*Sequential:* Only one processor at a time executes an atomic operation.

*Partially Synchronous:* A subset of the processors can simultaneously execute one atomic operation.

*Synchronous:* All processors simultaneously execute one atomic operation.

In our execution models every processor is always enabled and capable of executing an atomic operation. Thus without some notion of fairness, it would be possible for the scheduling discipline to let one processor execute continuously without causing a change in the configuration of the processor network.

The usual requirement is that a scheduler need not be fair, it need only be *proper*. When possible, a proper scheduler must schedule the next atomic operations in such a way that change at the configuration level is possible. That is, unless no alternative action is possible, the next atomic events scheduled for execution must result in a configuration different from the current one. In the case of randomized processor networks, the next configuration need only have a non-zero probability to be different.

But the requirement that a scheduler be proper has essentially no bearing on whether a protocol is correct or not. Its purpose is to permit the execution time of the protocol to be counted in terms of scheduling operations. Properties of the scheduler can be ignored completely if instead we define *time* in terms of configuration changes. If the scheduler schedules an operation, and as a result no configuration change occurs, then time has not passed.

A processor network can reach a configuration in which, no matter how the next atomic operations are scheduled, the next global configuration is identical to the current one. In this case we say that the processor network is *deadlocked*. Of course a network can be deadlocked at the configuration level yet individual processors can be changing nonobservable components of their states. When time is defined as above, a deadlocking protocol behaves like a halting program — once deadlocked, time ceases to pass.

## 3.3 Standard Models

The notions of operational atomicity and scheduling discipline combine to specify the four common computation models in the literature.

*Central Demon:* This is a single-cycle atomic processor network under the sequential scheduling discipline.

*Synchronous Demon:* This is a single-cycle atomic processor network under the synchronous scheduling discipline.

*Distributed Demon:* This is a single-cycle atomic processor network under the partially synchronous scheduling discipline.

*Read/Write Demon:* This is a read/write single atomic processor network under the partially synchronous scheduling discipline.

The above definitions are not strictly equivalent to those in the literature since the usual requirements are that the central, synchronous, and distributed demons be proper, and that the read/write demon be fair.

| Edge Name | Condition | Pictograph |
|---|---|---|
| oriented edge | $o(v) = i,\ o(w) \neq j$ | (→)——(→) |
| | $o(v) \neq i,\ o(w) = j$ | (←)——(←) |
| disoriented edge | $o(v) = i,\ o(w) = j$ | (→)——(←) |
| ignored edge | $o(v) \neq i,\ o(w) \neq j$ | (←)——(→) |

Figure 4.1: States of an edge $e = ((v, i), (w, j))$

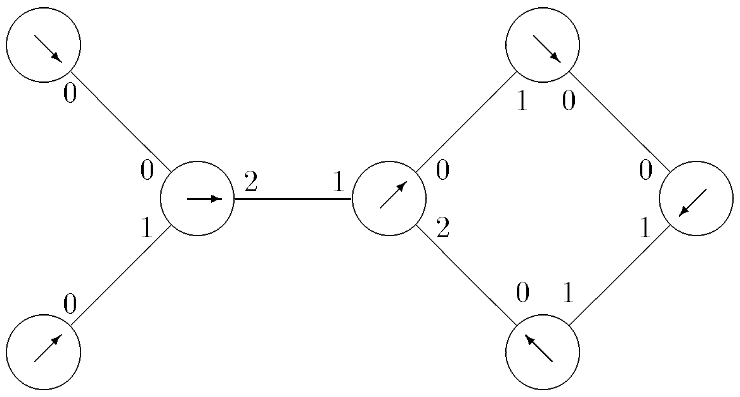

Figure 4.2: An oriented network

Also note that because all communications occur over edges, the only possible conflicting simultaneous atomic operations are a read and write to the same channel. Thus in the read/write single atomic model, partially synchronous scheduling is no more powerful than sequential scheduling in terms of the variety of behaviour of the processor network.

## 4 The Orientation Problem

To each vertex $v$ of degree $k$ is assigned an *orientation* $o(v) \in \{0, \ldots, k-1\}$. It is convenient to visualize each vertex as having an internal pointer whose head is directed toward port $o(v)$. The orientation of a vertex $v$ is generally a function of the state of the processor $P_v$. The orientation of the vertices at the ends of an edge induces a state for that edge, as defined in Figure 4.1. The orientation of a processor is clearly a component of the network configuration.

We say that a network is *oriented* iff every edge is oriented. That is, exactly one of the vertices at the ends of each edge is oriented toward that edge. There must be exactly as many vertices as edges for a network to be orientable, and thus such a network consists of a single ring with trees attached. Such an oriented network is illustrated in Figure 4.2.

In general terms, the *orientation problem* is to specify a protocol whose execution, given an initial configuration of orientations of vertices, eventually results in an oriented network. We say that a orientation protocol is *correct* under a given computation model if it orients every permissible initial configuration of every permissible network topology.

The property of being oriented can be either a *dynamic* property, or a *static* property. A network is dynamically oriented if it is oriented, yet the processor network continues to make configuration changes. A network is statically oriented if it is oriented and the processor network is deadlocked.

There are obvious scenarios in which the orientation problem is solved by the simple expedient of orienting the network at construction time. We are not interested in these. Nor are we interested in situations which allow protocols to use a dictator or elected leader to specify the orientation.

We are interested in *uniform* orientation protocols in which the processors have no ids and no idea of the global network topology. That is, the protocol computation by the processor at a vertex is a function only of the degree of the vertex.

By simple symmetry considerations (consider the ring of two processors), there is no correct uniform orientation protocol under any deterministic single-cycle atomic synchronous model. Any correct protocol will require randomization to break symmetry [IJ91].

## 5 Three Simple Random Protocols

Consider the following simple idea for an orientation protocol:

> A processor does nothing unless it is oriented towards a disoriented edge, in which case it randomly changes orientation.

For illustrative purposes, we consider using this idea to implement an orientation protocol in three models, each progressively less restricted in terms of model and scheduler. For simplicity, we restrict our attention to rings.

### 5.1 Single-Cycle Atomic, Synchronous Scheduling

First consider an execution model that is single-cycle atomic with a synchronous scheduling discipline. Call this the R1 protocol.

Each processor in the network is a simple randomized finite state machine. The channels between processors are simple direct connections. There is one type of processor in the network, possessing 2 ports, and the states $\{0, 1\}$. The state of a processor is directly identified with its orientation, with state $q$ indicating orientation toward port $q$.

The output ports continuously transmit a symbol from $\{H, T\}$, with the symbol currently transmitted to port $i$ being given by the port $i$ output function $\pi_i$ applied to the current state $q$.

$$\pi_i(q) = \begin{cases} H & i = q \\ T & i \neq q \end{cases}$$

Each port output function simply transmits whether the processor is oriented toward, $H$, or away from, $T$, the port.

The state transition function of a processor is

$$\delta(q) = \begin{cases} (0,1) & \text{if input port } q \text{ is } H \\ q & \text{otherwise} \end{cases}$$

where $(0,1)$ denotes the uniform random choice of either state 0 or 1 as the next state.

Define the configuration of the processor network to be the vector of states of each processor, and define an atomic operation in the obvious way as the steps consisting of reading the input ports, changing state, and updating the output ports.

It is clear that for this protocol, orientation is a static property of the network. We say that a protocol deadlocks if the network running the protocol deadlocks.

**Proposition 5.1** *The R1 protocol is deadlocked if and only if the network is oriented.*

To analyse the protocol, we introduce the higher level notion of an *interaction* between two processors, and will express the progress of the protocol in terms of interactions. For this reason, it is important that a processor be involved in at most one interaction at a time.

In the synchronous setting, an interaction consists of the simultaneous execution of an atomic operation by two processors at the end of a disoriented edge. Despite the fact that interactions are occurring simultaneously, they are independent, and we can analyse the protocol as if the interactions happened sequentially.

**Proposition 5.2** *If the R1 protocol is executed on a ring, starting in any configuration, then in $O(\text{size}(G)^2)$ expected number of interactions (with variance $O(\text{size}(G)^4)$) the protocol deadlocks (with the ring oriented).*

*Proof.* Arbitrarily assign an orientation to $G$ and call it *clockwise*. Thus each processor is oriented either clockwise or anti-clockwise. Consider the following statistic for $G$ given by

$$h(G) = \text{number of clockwise processors}$$

Then a network of size $n$ is oriented when $h(G) = n$ or $h(G) = 0$.

Now consider an interaction of two processors at a disoriented edge $e$. With probability $1/2$ this edge $(\rightarrow)$——$(\leftarrow)$ stays the same or becomes ignored $(\leftarrow)$——$(\rightarrow)$. In both cases $h(G)$ stays the same. With probability $1/4$ edge $e$ becomes $(\leftarrow)$——$(\leftarrow)$, and with probability $1/4$ it becomes $(\rightarrow)$——$(\rightarrow)$. Thus, with equal probability, $h(G)$ decreases or increases by 1 at every interaction.

Thus changes in $h(G)$ behave like a simple random walk on an interval with in which the probability of staying in the same place is $1/2$, and the probability of moving in either direction is $1/4$. The walk begins at a position somewhere on the interval $[0, n]$, and terminates when it hits either end of the interval.

The expected number of steps to a boundary is maximized when the initial position is in the middle. In such a situation, the expected number of steps before hitting either boundary is $n^2/2$ with a variance of $n^2(n^2+1)/6$. (See [Fel68] for the expected value. Some symbolic manipulation is required to compute the variance.)

Thus the expected number of interactions to orient $G$ is $O(n^2)$, with a variance of $O(n^4)$. $\square$

So long as the network is unoriented, the synchronous scheduling discipline ensures that at least one interaction is occurring at each configuration change, and thus the number of interactions is a loose upper bound on the execution time of the protocol. Since we make no assumptions about the initial configuration of the network the protocol is self-stabilizing. (An interesting problem is to compute a tighter bound on the time by considering the simultaneous interactions that occur in any synchronous step.)

## 5.2 Single-Cycle Atomic, Partially Synchronous Scheduling

Now suppose that we change to a partially synchronous scheduling discipline. In this case, interactions need not occur at each state transition, because both parties to an interaction are not required to execute simultaneously. In fact, interactions need not even occur, as in the case of

$$(\rightarrow)\!\!-\!\!\!-\!\!\!-\!\!(\leftarrow)\!\!-\!\!\!-\!\!\!-\!\!(\leftarrow)$$

where the processor in the middle can execute an arbitrarily large number of configuration changes before either of the end processors do even one.

To avoid this problem, the protocol implementation must ensure that the processors at the end of a disoriented edge are forced to interact. This is achieved by making the protocol self-synchronizing. That is, the parties to the interaction take turns waiting for each other to complete a phase of the interaction before changing state, thus preventing one processor from doing arbitrary numbers of state transitions. However, such a self-synchronizing protocol requires one of the pair of processors to begin the interaction — a symmetry which itself must be broken through randomization. Furthermore, the alternation created by the self-synchronization causes one processor to finish its participation in the interaction before the other processor. This permits the processor that finishes first to participate in a new interaction, while the other processor has yet to complete the original interaction — thus destroying the assumption of interaction independence, and the preceding analysis.

But we can preserve independence and also self-synchronize. This can be accomplished by preventing the case where both processors turn away (no change in the orientation statistic), combined with ensuring that the processor that acts last is the one that actually changes orientation. Thus the interaction must be complete before either processor can participate in a new interaction.

For the R2 protocol, we define the port alphabet to be $\Sigma = \{I, H, S, C\}$. The intuition behind the symbols is as follows: an $I$ at an input port means that the processor generating it is ignoring the processor receiving it; an $H$ at an input port means that the generating processor is oriented toward the receiving processor; a $S$ indicates that the sending processor will be staying in its current orientation; and a $C$ indicates the the sending processor will be changing orientation.

Each processor has the states $\{H_j, S_j, C_j \mid j \in \{0,1\}\}$. The state letter indicates the mode of the processor, and the subscript indicates its orientation. The port output functions simply transmit the mode of the processor to the port it is oriented toward, and send $I$ to the other ports. Thus

$$\pi_i(H_j) = \begin{cases} H & i = j \\ I & i \neq j \end{cases} \quad \pi_i(S_j) = \begin{cases} S & i = j \\ I & i \neq j \end{cases} \quad \pi_i(C_j) = \begin{cases} C & i = j \\ I & i \neq j \end{cases}$$

The state transition function $\delta$ is given by the following table, with dashes indicating no state change. (Note: $i \in \{0, 1\}$).

| | input from port $i \rightarrow$ | $H$ | $S$ | $C$ |
|---|---|---|---|---|
| | $H_i$ | $(S_i, C_i)$ | $C_i$ | – |
| current state | $S_i$ | – | $(S_i, C_i)$ | $H_i$ |
| | $C_i$ | $H_{(i+1) \bmod 2}$ | – | $(S_i, C_i)$ |

The key idea is that, unless both processors are in identical modes, only one processor is capable of making a transition that results in a configuration change.

When processors are in identical modes, they enter into an arbitration sequence which is broken by one processor entering the $S$ mode and the other entering the $C$ mode. The probability of the arbitration succeeding is 1/2, regardless of whether one processor or both are scheduled. The probability that the arbitration will require more than $k$ steps is $1/2^k$. It is simple to see that the expected number of arbitration steps (not configuration changes) is 2, with a variance of 2.

The fact the the protocol forces interacting processors to change states in a self-synchronizing way is easily verified by looking at the transition matrix above.

The off-diagonal entries correspond to the protocol operating on a disoriented edge — only one of the interacting processors can make a transition. The diagonal entries correspond to arbitration.

The possible reorientation outcomes for the protocol are equiprobable, regardless of how the processors are scheduled. Furthermore, the orientation statistic can now be analysed as a simple random walk. Also, this protocol will survive transient faults. Thus we have:

**Proposition 5.3** *If the R2 protocol is executed on a ring, starting in any configuration, then in $O(\text{size}(G)^2)$ expected number of interactions (with variance $O(\text{size}(G)^4)$) the protocol deadlocks with the ring oriented.*

## 5.3 Read/Write Single Atomic, Partially Synchronous Scheduling

Finally, we consider the weaker model in which the basic instruction cycle consists of a read or write followed by a state change. This is illustrated in Figure 5.1. The current state of the machine is $q$. The function $\rho$ specifies the port to be read or written, and $\omega$ specifies the value to be written.

Because transient faults can change the contents of the port registers without a processor's knowledge, they must be continuously refreshed. Thus the basic cycle of the protocol consists of a read from a port followed by a sequence of writes to ports.

```
while ( 1 ) {
    Read $v$ from port $\rho(q)$ and change state $q$ to $\delta(q, v)$
    or
    Write value $\omega(q)$ to port $\rho(q)$ and change state $q$ to $\delta(q)$

}
```

Figure 5.1: Basic Read/Write Single Atomic execution cycle.

Since the read and write to a port is not packaged into a single atomic operation it becomes much more difficult for one processor to reliably determine the state of the one it is communicating with. For example, the simple method of arbitration used above where processors randomly chose between two values until they each obtain different ones will not always terminate under a partially synchronous scheduler. As pointed out by Amos Israeli (private communication) the scheduler can manipulate the executions of the two processors in such a way that they always think they have the same values and thus they continue to arbitrate.

The simplest way to cope with the wide range of possible execution sequences of two interacting processors is to impose some kind of notion of atomic interaction onto their behaviour. This can be done by forcing every pair of interacting processors to act as if they were synchronous single-cycle atomic machines. That is, one processors reads from a port never overlap with the other processors write to the port.

We do this by running a low-level deterministic protocol whose only purpose is to keep every pair of processors in close synchronization. Each processor maintains a synchronization state for each port, and actually executes the protocol only when it is paying attention to the port. For a single port, the synchronization protocol has four states, and is described by the following transition table:

| | port input → | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| | 0 | 1 | 1 | 0 | 0 |
| current state | 1 | 1 | 2 | 2 | 0 |
| of port | 2 | 0 | 2 | 3 | 3 |
| | 3 | 0 | 0 | 3 | 0 |

As two processors execute this protocol, the states of their communicating ports can never differ by more than 1 (modulo 4), except at startup or under a transient fault. The 0 entries serve the purpose of resynchronizing the processors in the event of a transient fault. Once synchronized, the protocol advances through the port states in sequence.

We can now implement the R2 protocol on top of the synchronization protocol. We map the synchronization states onto four phases: idle (phase 0), read input (phase 1), idle (phase 2), write output (phase 3). The synchronization states are now used to insure that any actions taken during the read input phase of one processor cannot overlap actions taken during the other processor's write output phase. To transmit the four symbols of the R2 protocol requires a port alphabet of 16 symbols $\{I, H, S, C\} \times \{0, 1, 2, 3\}$, and a machine of 96 states ($4 \times 4 \times 6$, two synchronization protocols and the R2) constructed in the obvious way by combining the R2 and synchronization protocols. Call the resulting not so simple protocol R3.

Since the synchronization protocol is deterministic, it does not affect the probabalistic analysis of the correctness or performance of the R2 protocol, and so we have:

**Proposition 5.4** *If the R3 protocol is executed on a ring, starting in any configuration, then in $O(\text{size}(G)^2)$ expected number of interactions (with variance $O(\text{size}(G)^4)$) the protocol deadlocks with the ring oriented.*

In all three protocols, the network configuration is the vector of orientations of all the processors, and every interaction is composed of a constant or expected constant number of configuration changes. Thus, interactions are a reasonable measure of execution time. We wish to emphasize that an interaction at the configuration level may involve a number of protocol atomic actions, each of which consists of numerous changes of a processor state.

## 6 The High-Level Salad Protocol

So far, we have obtained three orientation protocols in three progressively weaker models, each with the same expected number of interactions as the original Israeli-Jalfon protocol. Their main weakness is their high variance, and that the analysis only works on a ring.

We now use the ideas of the salad passing protocol to give an orientation protocol that works in the weakest atomic model and strongest scheduler, and that generalizes to unicyclic graphs.

We will present our orientation protocol in two stages. The high-level protocol will be specified in terms of normal finite state machines, making the additional assumption that it is possible to arbitrate certain kinds of conflicts between adjacent processors. We will show that this protocol is correct and has the claimed performance. Then we will show how to implement the high-level protocol with randomized finite state machines.

The high-level protocol corresponds roughly to the salad protocol described in Figure 2.1. In addition to an orientation as described above, each processor has a *mode* which is either $P$ or $W$. We denote the mode of the vertex $v$ by $m(v)$. The mode $P$ stands for *passing mode*, which can be thought of as the processor being in possession of an extra salad and wanting to pass it to the neighbour it is oriented towards. The mode $W$ stands for *waiting mode*, which can be thought of as the processor waiting for a salad to appear.

This induces a state for each edge described by the orientation and mode of the processors at its ends. Figure 6.1 defines the various states of an edge.

We note that because the protocol must be self-stabilizing, we cannot make any assumptions about the initial state of the network. The only thing that we can assume

| Edge Name | Condition | Pictograph |
|---|---|---|
| properly oriented edge | $o(v) = i,\ m(v) = W,\ o(w) \neq j$ | $(\overset{W}{\rightarrow})$——$(\rightarrow)$ |
| improperly oriented edge | $o(v) = i,\ m(v) = P,\ o(w) \neq j$ | $(\overset{P}{\rightarrow})$——$(\rightarrow)$ |
| properly disoriented edge | $o(v) = i,\ m(v) \neq m(w),\ o(w) = j$ | $(\overset{P}{\rightarrow})$——$(\overset{W}{\leftarrow})$ |
| improperly disoriented edge | $o(v) = i,\ m(v) = m(w),\ o(w) = j$ | $(\overset{P}{\rightarrow})$——$(\overset{P}{\leftarrow})$<br>or $(\overset{W}{\rightarrow})$——$(\overset{W}{\leftarrow})$ |
| ignored edge | $o(v) \neq i,\ o(w) \neq j$ | $(\leftarrow)$——$(\rightarrow)$ |

Figure 6.1: States of an edge $e = ((v,i),(w,j))$

| Current Configuration | Next Configuration |
|---|---|
| $o(v) = i,\ m(v) = P$<br>$o(w) = j,\ m(w) = W$<br><br>$(\overset{P}{\rightarrow})$——$(\overset{W}{\leftarrow})$ | $o(v) = i,\ m(v) = W$<br>$o(w) = (j+1) \bmod \deg(w),\ m(w) = P$<br><br>$(\overset{W}{\rightarrow})$——$(\overset{P}{\rightarrow})$ |
| $o(v) = i,\ o(w) = j$<br>$m(v) = m(w)$<br><br>$(\overset{P}{\rightarrow})$——$(\overset{P}{\leftarrow})$ or $(\overset{W}{\rightarrow})$——$(\overset{W}{\leftarrow})$ | $o(v) = i,\ o(w) = j$<br>$m(v) \neq m(w)$ (arbitrarily)<br><br>$(\overset{W}{\rightarrow})$——$(\overset{P}{\leftarrow})$ or $(\overset{P}{\rightarrow})$——$(\overset{W}{\leftarrow})$ |

Figure 6.2: The high level protocol at edge $e = ((v,i),(w,j))$

is that there is sufficient time between transient faults for the network to stabilize. We call this interval between transient faults an *execution.*

For the high level protocol, we assume that processors make changes in orientation and mode instantaneously like normal finite state machines. Processors will only interact with the neighbour they are oriented toward, and thus the rules for the protocol are very simple: *all progress toward orientation occurs at disoriented edges.* (At the low level, writes may occur to ports other than the one the processor is oriented toward.)

Suppose that $e = ((v,i),(w,j))$ is a disoriented edge. The protocol at $e$ is described by Figure 6.2. Each application of the protocol to a current configuration of two adjacent processors which produces a next configuration of the two processors is termed an *interaction.* If no interactions are possible on any edge, then the network is *deadlocked.*

The behaviour of the protocol depends on whether the edge is properly or improperly disoriented. For a properly oriented edge, the passing processor drops into waiting mode, and the waiting processor re-orients itself to its next port and enters passing mode. On the other hand, for an improperly oriented edge, the mode conflicts must first be *arbitrated,* converting the edge into a properly oriented one. We assume only that some arbitration mechanism exists. It need not be fair. (In the implementation, this arbitration will be done randomly.)

The following is a direct consequence of the definition of the high-level protocol.

**Proposition 6.1** *A network executing the high-level protocol is deadlocked if and only if it is oriented.*

## 6.1 Correctness of the high-level protocol

We must prove that every possible configuration of the network eventually deadlocks. How does the protocol make progress? Consider a possible execution of the protocol on a path:

$$\begin{array}{ll}
0: & \underline{\qquad}\binom{P}{\rightarrow}\underline{\;e_1\;}\binom{W}{\leftarrow}\underline{\;e_2\;}\binom{W}{\leftarrow}\underline{\;e_3\;}\binom{W}{\rightarrow}\underline{\;e_4\;}\binom{W}{\leftarrow}\underline{\qquad} \\
1: & \underline{\qquad}\binom{W}{\rightarrow}\underline{\;e_1\;}\binom{P}{\rightarrow}\underline{\;e_2\;}\binom{W}{\leftarrow}\underline{\;e_3\;}\binom{W}{\rightarrow}\underline{\;e_4\;}\binom{W}{\leftarrow}\underline{\qquad} \\
2: & \underline{\qquad}\binom{W}{\rightarrow}\underline{\;e_1\;}\binom{W}{\rightarrow}\underline{\;e_2\;}\binom{P}{\rightarrow}\underline{\;e_3\;}\binom{W}{\rightarrow}\underline{\;e_4\;}\binom{P}{\leftarrow}\underline{\qquad} \\
3: & \underline{\qquad}\binom{W}{\rightarrow}\underline{\;e_1\;}\binom{W}{\rightarrow}\underline{\;e_2\;}\binom{P}{\rightarrow}\underline{\;e_3\;}\binom{P}{\leftarrow}\underline{\;e_4\;}\binom{W}{\leftarrow}\underline{\qquad} \\
4: & \underline{\qquad}\binom{W}{\rightarrow}\underline{\;e_1\;}\binom{W}{\rightarrow}\underline{\;e_2\;}\binom{W}{\rightarrow}\underline{\;e_3\;}\binom{P}{\leftarrow}\underline{\;e_4\;}\binom{W}{\leftarrow}\underline{\qquad} \\
5: & \underline{\qquad}\binom{W}{\rightarrow}\underline{\;e_1\;}\binom{W}{\rightarrow}\underline{\;e_2\;}\binom{P}{\leftarrow}\underline{\;e_3\;}\binom{W}{\leftarrow}\underline{\;e_4\;}\binom{W}{\leftarrow}\underline{\qquad} \\
6: & \underline{\qquad}\binom{W}{\rightarrow}\underline{\;e_1\;}\binom{P}{\leftarrow}\underline{\;e_2\;}\binom{W}{\leftarrow}\underline{\;e_3\;}\binom{W}{\leftarrow}\underline{\;e_4\;}\binom{W}{\leftarrow}\underline{\qquad}
\end{array}$$

We can think of the protocol as transferring the disorientation of a edge (eg. $e_1$) to an adjacent edge (eg. $e_2$), leaving the first edge oriented. The disorientation state keeps moving in its original direction until it either collides with an ignored edge (eg. $e_3$ at step 2), or reflects off of an improperly disoriented edge (eg. $e_3$ at step 3). A collision with an ignored edge reduces the number of unoriented edges and so the protocol makes progress. Reflection off of an improperly disoriented edge sometimes makes progress in orientation, and always ensures that arbitration is never required again at that edge.

In addition to improperly oriented edges at the beginning of execution (eg. $e_4$), the reorientation of processors during execution can create improperly oriented edges (eg. $e_3$ at step 3). However, the definition of the protocol ensures that:

**Lemma 6.2** *During the execution of the protocol, at most one arbitration can occur at each edge.*

This observation is important because it means that there is an upper bound on the number of reflections that can occur at an edge. Ignored edges are also important as they are points at which orientation conflicts are resolved.

**Lemma 6.3** *The execution of the protocol cannot generate any ignored edges.*

It is convenient to reason about the orientation protocol's behaviour on a tree. An *edge-rooted tree* $T$ with root vertex $v$ and root edge $e$ is constructed by taking a tree with root $v$ and edge $e$ incident on $v$, and deleting the vertex at the other end of $e$. All of our trees will be edge-rooted, so we simply use the term *tree*. Figure 6.3 illustrates a prototypical tree used in the proofs that follow.

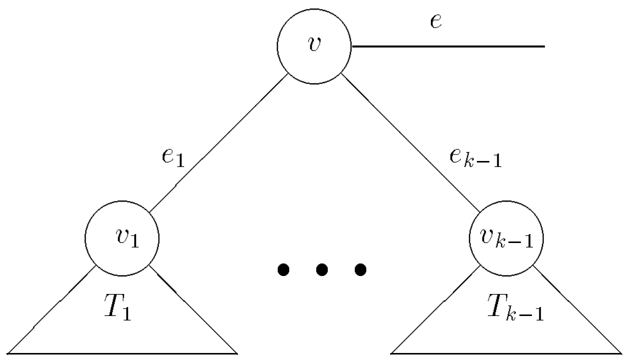

Figure 6.3: A typical edge-rooted tree.

The next two lemmas show that ignored and disoriented edges are balanced in a network. Let $\#_I(G)$ and $\#_D(G)$ denote, respectively, the number of ignored and disoriented edges in the network $G$. When applied to a tree $T$, the root edge of $T$ is not counted.

**Lemma 6.4** *Let $T$ be a tree with root vertex $v$ and root edge $e$. If $v$ is oriented toward $e$ then $\#_I(T) = \#_D(T)$. If $v$ is oriented away from $e$ then $\#_I(T) = \#_D(T) - 1$.*

*Proof.* We proceed by induction on the size of $T$. For the case of $T$ being a single vertex, $v$ must be oriented toward $e$, and we have $\#_I(T) = \#_D(T) = 0$.

Suppose that vertex $v$ has degree $k > 1$, and edges $e_1, \ldots, e_{k-1}$ in addition to $e$. Then $T$ looks like the tree of Figure 6.3.

When $v$ is oriented toward $e$, each of the edges $e_l$, $1 \leq l < k$, is either ignored or oriented. If $e_l$ is oriented, then $v_l$ is oriented toward $e_l$, and by induction $\#_I(T_l) = \#_D(T_l)$. If $e_l$ is ignored, then $v_l$ is oriented away from $e_l$, and by induction $\#_I(T_l) = \#_D(T_l) - 1$. Adding the ignored edge $e_l$ maintains the balance between ignored and disoriented edges.

When $v$ is oriented away from $e$, then it is oriented toward exactly one edge $e_l$ which is either oriented or disoriented. Balance is maintained for the other subtrees as above. If $e_l$ is oriented, then $v_l$ is oriented away from $e_l$, and by induction $\#_I(T_l) = \#_D(T_l) - 1$, and so $\#_I(T) = \#_D(T) - 1$. If $e_l$ is disoriented, then $v_l$ is oriented toward $e_l$, and by induction $\#_I(T_l) = \#_D(T_l)$. Accounting for the disoriented $e_l$ we have $\#_I(T) = \#_D(T) - 1$. $\square$

**Lemma 6.5** *Let $G$ be a unicyclic network. Then $\#_I(G) = \#_D(G)$.*

*Proof.* If $G$ is oriented then $\#_I(G) = \#_D(G) = 0$. Suppose that $G$ is not oriented. Pick any edge $e = ((v,i),(w,j))$ on the cycle of $G$ and cut it, attaching a leaf $u$ to vertex $w$ with the edge $f = ((u,0),(w,j))$. The net result is a tree $T$ with root vertex $v$ and root edge $e$.

If $e$ was originally disoriented, then edge $f$ will be disoriented, and $v$ will be directed toward $e$. That is, edge $e$ : $(\rightarrow)\rule{2em}{2pt}(\leftarrow)$ becomes the edge $f$ : $(\rightarrow)\rule{2em}{2pt}(\leftarrow)$, and $e$ becomes the root edge $(\rightarrow)\rule{2em}{2pt}$. Applying Lemma 6.4 to $T$ we have $\#_I(T) = \#_D(T)$. Since the disoriented edge $f$ in $T$ accounts for the originally disoriented edge $e$, we have balance for $G$.

If $e$ was originally ignored, then edge $f$ will be oriented, and $v$ will be directed away from $e$. That is, edge $e:\ (\leftarrow)$——$(\rightarrow)$ becomes the edge $f:\ (\rightarrow)$——$(\rightarrow)$, and $e$ becomes the root edge $(\leftarrow)$——. Applying Lemma 6.4 to $T$ we have $\#_I(T) = \#_D(T) - 1$. The oriented edge $f$ is not counted in $T$, nor was the originally ignored $e$, so we have balance for $G$.

If $e$ was originally oriented, we have one of the above cases. □

Since progress is made as disoriented edges move about the tree, we need to know how they can interact. In a tree $T$ we say that an edge $e$ is *between* edges $e_1$ and $e_2$ if $e$ lies on a path between $e_1$ and $e_2$. We say that a disoriented edge $e_1$ in tree $T$ is *covered* if there exists an ignored edge between $e_1$ and the root edge of $T$.

**Lemma 6.6** *Let $T$ be a tree with root vertex $v$ and root edge $e$. If $v$ is oriented toward $e$ then every disoriented edge in $T$ is covered. If $v$ is oriented away from $e$ then all but one disoriented edge in $T$ is covered.*

*Proof.* We proceed by induction on the size of $T$. For the case of $T$ being a single vertex, we have no disoriented edges.

Suppose that vertex $v$ has degree $k > 1$, and edges $e_1, \ldots, e_{k-1}$ in addition to $e$. Then $T$ looks like the tree of Figure 6.3.

When $v$ is oriented toward $e$, each of the edges $e_l$, $1 \leq l < k$, is either ignored or oriented. If $e_l$ is oriented, then $v_l$ is oriented toward $e_l$, and by induction all disoriented edges of $T_l$ are covered. If $e_l$ is ignored, then $v_l$ is oriented away from $e_l$, and by induction $T_l$ has one uncovered disoriented edge $f$. But $e_l$ is between $f$ and the root edge $e$, and so $f$ is covered. Thus all disoriented edges in $T$ are covered.

When $v$ is oriented away from $e$, then it is oriented toward exactly one edge $e_l$ which is either oriented or disoriented. Any uncovered disoriented edges for the other subtrees are covered as above. If $e_l$ is oriented, then $v_l$ is oriented away from $e_l$, and by induction $T_l$ has an uncovered disoriented edge which remains uncovered in $T$. If $e_l$ is disoriented, then $v_l$ is oriented toward $e_l$, and by induction $T_l$ has all disoriented edges covered. Thus $e_l$ is the only uncovered edge of $T$. □

Ignored edges serve as separators between disoriented edges in the following manner.

**Corollary 6.7** *Let $T$ be a tree with root vertex $v$ and root edge $e$ with $v$ oriented away from $e$. Then $T$ can be partitioned into $1 + \#_I(T)$ subtrees such that the root edge of each subtree corresponds to an ignored edge in $T$, and every subtree contains exactly one disoriented edge.*

Now consider how a single proper disorientation moves about a tree of otherwise properly oriented edges. Note that the root vertex is oriented away from the root edge. The protocol forces the disoriented edge to move about the tree in a depth first order induced by the port numbers at each vertex. For the typical tree (Figure 6.3), suppose that $e_1$ is properly disoriented with $v$ in passing mode $P$. The disorientation moves from $e_1$ into subtree $T_1$, moves about $T_1$ in depth first order, and returns to $e_1$ with $v_1$ in mode

$P$ and $v$ in waiting mode $W$. The disorientation then passes to $e_2$. This process continues until $v$ is oriented toward $e$ in mode $P$. That is, the disorientation has moved out of $T$. We call the sequence of edges that a disorientation follows as it depth first searches the tree a *trip*.

Two things can affect the trip that a disorientation takes in an arbitrary tree. One is encountering an ignored edge. When this happens the protocol replaces the ignored and disoriented edges with properly oriented ones, and the trip terminates. The other thing that can occur is for the disorientation to encounter an improperly oriented edge (eg. $e_4$ step 2 of our example). In this case, it is possible for the resulting arbitration to cause the disoriented edge to bounce, causing the subtree below to be skipped (when approaching from above), or the subtree to be traversed again (when approaching from below). A bounce, since it requires an arbitration, can occur at most once at each particular edge. Call an edge which has not yet participated in an arbitration an *unarbitrated edge*.

Thus we can measure progress in the protocol by observing the decrease in the number of ignored and unarbitrated edges.

**Lemma 6.8** *Let $T$ be a tree with root vertex $v$ and root edge $e$. Then (1) $T$ contains only oriented edges; or (2) every oriented edge in $T$ is properly oriented, and there is exactly one disoriented edge; or (3) in at most $2\,\mathrm{size}(T)$ interactions between processors of $T$ the total number of ignored plus unarbitrated edges in $T$ will decrease by 1.*

*Proof.* In order for the protocol to be active, $T$ must contain at least one disoriented edge, so we assume that (1) does not hold. If there are no ignored edges in $T$, then arbitrations will only occur if some edges are improperly oriented, so we also assume that (2) does not hold. Then $T$ can contain exactly one improperly disoriented edge; or exactly one properly disoriented edge and some improperly oriented edges; or some ignored or some disoriented edges.

Any interactions that occur in $T$ are at disoriented edges, and these cause each such edge to progress along its depth first trip through $T$.

If there is exactly one disoriented edge, and it is improper, then an arbitration will occur at the edge to turn it into a properly disoriented one, thus decreasing the number of unarbitrated edges by 1.

If there is exactly one properly disoriented edge then in at most $2\,\mathrm{size}(T)$ interactions the disorientation must encounter an improperly oriented edge and cause an arbitration. Note, that the disorientations could move out of $T$ and a new one enter — interactions are all that is important.

The final case occurs when there is more than one disoriented edge. By Lemma 6.4 there must be at most 1 more disoriented than ignored edges in $T$. By Corollary 6.7, the motions of the disoriented edges are occurring in disjoint portions of $T$ connected by ignored edges. The only way that a disorientation can miss an ignored edge is for it to bounce off of an unarbitrated, improperly oriented edge, which results in an arbitration. If this does not happen, at most $2\,\mathrm{size}(T)$ interactions are required before one of the disoriented edges cancels with an ignored edge.

In all cases the number of ignored plus unarbitrated edges is reduced by 1. $\square$

**Corollary 6.9** *In at most* $O(\mathrm{size}(T)^2)$ *interactions within tree* $T$ *with root vertex oriented away from the root edge, exactly one edge is properly disoriented and all other edges are properly oriented.*

*Proof.* Since resolving an ignored edge can require an arbitration, the number of ignored plus unarbitrated edges is bounded by $2\,\mathrm{size}(T)$. New ones are never created. ☐

**Lemma 6.10** *Let* $G$ *be a unicyclic network. Then the protocol deadlocks on* $G$*.*

*Proof.* We proceed by induction on the size of $G$, and suppose that the claim holds for all networks of smaller size.

Suppose for contradiction that some particular execution of the protocol does not deadlock on $G$. Then there is at least one disoriented edge in $G$ and by Lemma 6.5 there is an equal number of ignored edges.

Since ignored edges are never created, there must be an ignored edge $e = ((v, i), (w, j))$ that existed at the beginning of the execution and that will exist forever. So the processors at both ends of $e$ never orient towards $e$.

Suppose that the edge $e$ is on the cycle of $G$. We cut the network at edge $e$, and add a leaf vertex $u$ with edge $((u, 0), (w, j))$ to create a tree $T$ with root vertex $v$ and root edge $e$. Since neither $v$ nor $w$ orient towards edge $e$, the particular execution of the protocol must also fail to deadlock when projected onto $T$.

But, since vertex $v$ is oriented away from $e$, by Corollary 6.9, eventually $T$ will contain one properly disoriented edge, and all others will be properly oriented. This properly disoriented edge must eventually move towards $e$, and so $e$ cannot remain ignored. This contradicts the choice of $e$.

Thus $e$ must be inside a tree. It must connect a subtree $T_w$ of size at least 2 to the rest of $G$. ($T_w$ cannot be a leaf because then $e$ would not be ignored.) So we can cut the network at edge $e$, and add a leaf vertex $u$ with edge $((u, 0), (w, j))$ to create a new network $G'$. Vertex $w$ never orients toward this new edge, so the particular execution of the protocol behaves the same when projected onto $G'$, and so must not deadlock. But $G'$ is smaller than $G$ and so this contradicts the inductive assumption.

Thus the protocol always deadlocks on $G$. ☐

**Corollary 6.11** *Let* $G$ *be a unicyclic network. Then in at most* $O(\mathrm{size}(G)^2)$ *interactions between processors of* $G$ *the protocol deadlocks.*

*Proof.* Consider a possible serialization of a protocol execution on $G$, and consider the edge $e$ of the cycle of $G$ that remained ignored for the longest time. The proof of Corollary 6.9 shows that this $e$ could have remained ignored for at most $O(\mathrm{size}(G)^2)$ interactions. So after these interactions, no edges of the cycle are ignored, and since there are exactly as many processors as edges on the cycle, the cycle is oriented.

Any remaining ignored edges occur in subtrees of $G$, and further interactions cannot involve processors on the cycle of $G$, so these interactions are confined to subtrees.

Consider a subtree $T$, and its ignored edge $e$ closest to the root. By Corollary 6.9, for the subtree $S$ with root edge $e$, in $O(\text{size}(S)^2)$ interactions within $S$, every edge of $S$ is properly oriented except for one properly disoriented edge $f$. By Corollary 6.7, edge $f$ is covered by the ignored edge $e$, and in $O(\text{size}(T))$ interactions they will cancel.

Thus at most $O(\text{size}(G)^2)$ interactions occur before the protocol deadlocks. □

Thus we have:

**Proposition 6.12** *If the unicyclic network $G$ is started in any state, and no transient faults occur, then in $O(\text{size}(G)^2)$ interactions the protocol deadlocks with the network oriented.*

## 7 The low-level salad protocol

We now show how to implement the salad protocol under the single-cycle atomic model with a partially synchronous scheduler. It is then a simple matter to use the techniques of the R3 protocol to obtain an implementation under the read/write single atomic model with a partially synchronous scheduling discipline.

The processors in the salad protocol are like those in R2, except that a processor at a degree $k$ vertex has $k$ ports.

When processor $P_v$ is in state $q_v$, the symbol transmitted to output port $i$ of $P_v$ is $\pi_i(q_v)$ as given by the processor's port output functions.

For our protocol, we define the port alphabet to be $\{I, W, P, R\}$. The intuition behind the symbols is as follows: an $I$ at an input port means that the processor generating it is ignoring the processor receiving it; a $W$ ($P$, $R$) at an input port means that the generating processor is waiting to receive, (willing to pass, just received the pass).

Each $k$-port processor has the states $\{W_j, P_j, R_j \mid 0 \leq j < k\}$. The state letter is identified with the mode of the processor, and the subscript with the orientation of the processor. The port output functions simply transmit the mode of the processor to the port it is oriented toward, and send $I$ to the other ports. Thus

$$\pi_i(W_j) = \begin{cases} W & i = j \\ I & i \neq j \end{cases} \quad \pi_i(P_j) = \begin{cases} P & i = j \\ I & i \neq j \end{cases} \quad \pi_i(R_j) = \begin{cases} R & i = j \\ I & i \neq j \end{cases}$$

The state transition function $\delta$ is given by the following table. It has $6k$ entries for a $k$-port machine. Note how the state transition function ignores any ports other then the one that the processor is currently oriented toward.

| | input from port $i \rightarrow$ | $W$ | $P$ | $R$ |
|---|---|---|---|---|
| | $W_i$ | $(W_i, R_i)$ | $R_i$ | $-$ |
| current state | $P_i$ | $-$ | $(W_i, R_i)$ | $W_i$ |
| | $R_i$ | $P_{(i+1) \bmod k}$ | $-$ | $(W_i, R_i)$ |

The key idea is that this low-level protocol is self-synchronizing. At each properly disoriented edge exactly one of the interacting processors is capable of making a transition, and all future interactions between the two processors remain synchronized (barring transient faults).

At a properly disoriented edge, the low-level protocol enters an arbitration sequence which is broken by the edge becoming properly disoriented. As for the R2 protocol, the expected time to arbitrate is 2 with a variance of 2. To ensure that two interacting processors can always progress from any state, we require the $R$ vs $R$ transition. Under normal functioning of the protocol such a situation will never occur.

The fact the the protocol forces interacting processors to change states in a self-synchronizing way is easily verified by looking at the transition matrix.

The off-diagonal entries correspond to the protocol operating on a properly disoriented edge — only one of the interacting processors can make a transition. The diagonal entries correspond to arbitration. Processors can change state without being synchronized until they are arbitrated into a proper disorientation. (It is worth comparing this to the R2 protocol, in which arbitrations can occur repeatedly at an edge.)

An interaction of the high-level protocol thus corresponds to $O(1)$ expected configuration changes (variance $O(1)$) in the low-level protocol, and we have that

**Proposition 7.1** *Any unicyclic network $G$ executing the single-cycle atomic low-level protocol will self-stabilize into a deadlocked, oriented configuration in an expected time of $O(\text{size}(G)^2)$.*

Finally, we obtain:

**Theorem 7.2** *Under the read/write single atomic model with partially synchronous scheduler there exists a uniform orientation protocol that will self-stabilize on any unicyclic network (G) into a deadlocked, oriented configuration within $O(\text{size}(G)^2)$ expected number of configuration changes with variance $O(\text{size}(G)^2)$.*

We conjecture that $O(\text{size}(G)^2)$ is also the lower bound for this problem.

## 8 Acknowledgements

We wish to thank Amos Israeli for showing us that the problem was interesting, and Joe Culberson for much stimulating discussion and catching subtle bugs in our earlier versions. Wayne Eberly is responsible for the key idea of the proof for Proposition 5.2.

## References

[DIJ90] A. Dolev, A. Israeli, and M. Jalfon. Self-stabilization of dynamic systems. In 9th *Ann. ACM Symp. on Principles of Distributed Computation*, pages 103–118. Association for Computing Machinery, August 1990.

[Fel68] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, New York, third edition, 1968.

[IJ91] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 1991. To appear.

[Lam86a] L. Lamport. The mutual exclusion problem: Part i—a theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[Lam86b] L. Lamport. The mutual exclusion problem: Part ii—statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.