Theory Comput. Systems **37**, 585–607 (2004)
DOI: 10.1007/s00224-004-1080-z

Theory of Computing Systems

Compact Routing Schemes for Dynamic Ring Networks*

Danny Krizanc,[1] Flaminia L. Luccio,[2] and Rajeev Raman[3]

[1]Mathematics Department, Wesleyan University,
Middletown, CT 06459, USA
dkrizanc@wesleyan.edu

[2]Dipartimento di Scienze Matematiche, Università degli Studi di Trieste,
34127 Trieste, Italy
luccio@dsm.univ.trieste.it

[3]Department of Mathematics and Computer Science, University of Leicester,
Leicester LE1 7RH, England
R.Raman@mcs.le.ac.uk

Abstract. We consider the problem of routing in an asynchronous dynamically changing ring of processors using schemes that minimize the storage space for the routing information. In general, applying static techniques to a dynamic network would require significant re-computation. Moreover, the known dynamic techniques applied to the ring lead to inefficient schemes. In this paper we introduce a new technique, Dynamic Interval Routing, and we show tradeoffs between the stretch factor, the adaptation cost, and the size of the update messages used by routing schemes based upon it. We give three algorithms for rings of maximum size N: the first two are deterministic, one with adaptation cost zero but worst case stretch factor $\lfloor N/2 \rfloor$, the other with worst case adaptation cost $O(N)$ update messages of $O(\log N)$ bits and stretch factor 1. The third algorithm is randomized, uses update messages of size $O(k \log N)$, has adaptation cost $O(k)$, and expected stretch factor $1 + 1/k$, for any integer $k \geq 3$. All schemes require $O(\log N)$ bits per node for the routing information and all messages headers are of $O(\log N)$ bits.

* This research was supported by an NSERC grant and by MIUR progetto "Matematica per le scienze e la tecnologia" Università di Trieste. Rajeev Raman's work was supported in part by EPSRC Grant GR/L92150. A preliminary version of this paper was presented at the IEEE 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP '99) [17].

1. Introduction

The design of routing schemes that minimize the space devoted to routing tables in a network is an active area of research [3], [7], [10]–[16], [18]–[21]. Most of the research done in this area has concerned static networks and has focused on the tradeoffs between the space required for routing tables and the quality of the routing paths the tables defined. In general, to apply such static compact routing techniques to a dynamic network it would be necessary to perform a global re-computation of all the routing tables in the network. A much better approach is to consider schemes that require only a limited number of table updates (in the worst case or in an amortized sense) whenever a change occurs in the network [1], [2], [4]–[6], [8], [9].

A *static* routing scheme is composed of distributed routing tables (one at each node) and a routing procedure which uses the routing tables to perform the message delivery. A *dynamic* routing scheme consists also of a distributed update procedure which updates the routing tables whenever a change occurs in the network. In this paper we assume the changes include processors going off-line or coming on-line, in a fault-free manner, as is the case when, e.g., users are logging in or out, processors are being taken off-line for maintenance, new processors are being added to the network, etc. (In particular, we assume that processors going off-line complete the update procedure first.) In the worst case a single change may require that all the routing tables in the network have to be updated. It is desirable to design dynamic routing schemes that limit the amount of updating that must occur per change. We are interested in finding tradeoffs between the length of the routing paths, the space requirements of the routing tables, and the worst case or amortized number of messages exchanged per topology change, for dynamic routing schemes.

An important example of static routing schemes are k-Interval Routing Schemes (k-IRSs) [18], [21]: In an N node network, every node is labeled with a different value in the set $\{0, \ldots, N-1\}$; every arc e_i leaving a node i is assigned a set of at most k disjoint intervals $[a_i^l, b_i^l]$, $l = 1, \ldots, s$, $s \leq k$, such that $a_i^l, b_i^l \in \{0, \ldots, N-1\}$ and such that every node in G is in precisely one of the intervals assigned to an arc leaving i. Messages from i to j are forwarded through the arc labeled with the interval containing j. Interval routing schemes are an example of compact routing schemes. Note that the space required to store the routing table of a single node for a k-IRS is $O(kd \log N)$ bits where d is the degree of the node. For small k and d this is a significant saving over the complete table which requires $O(N \log d)$ bits per node in the worst case.

As an example of a dynamic routing scheme we introduce Dynamic Interval Routing Schemes (DIRSs). A DIRS for a network with maximum size N is based on the 1-IRSs (shortly IRSs): nodes are labeled by distinct values in the set $\{0, \ldots, N-1\}$ and arcs are labeled by disjoint intervals of values in the same set. However, not all of the processors may be on-line at all times, i.e., intervals may contain the label of processors that are not on-line. As changes may occur in the network, an update procedure is defined in order to modify the routing tables dynamically, i.e., the range of the intervals assigned to the arcs.

We require the following definitions. A processor is said to be *pending* if it has come on-line or is going off-line but has not yet completed the update procedure associated with the change it has caused in the network. After completing the update, the processor is said to be *active* if it comes on-line and *non-active* if it goes off-line. We say that

the system has reached *quiescence* if there are no topological updates pending, i.e., all processors are either active or non-active (see [1]).

We say that the system delivers all messages correctly if:

1. A message travels only a bounded number of steps.
2. A receiver receives a message if it is active during the entire lifetime of the message.

We assume that the communication between two neighboring processors that are active or pending has cost 1. We consider the following complexity measures:

1. The *space complexity* for the routing scheme, i.e., the maximum number of bits stored in each processor for the routing information during the quiescent state.
2. The *update message size*, i.e., the size of the update messages in bits.
3. The *adaptation cost*, i.e., the number of update messages generated per insertion and deletion of processors (this cost is worst case or amortized over the number of processors that go on-line and off-line in the system).
4. The *stretch factor* for the routing scheme, i.e., the maximum ratio between the length of the routing path between any two active processors and the length of a shortest path between them. The stretch factor is computed only when the destination processor is active and when the system has reached a state of quiescence, otherwise it can trivially be shown to be $\Omega(N)$ [1].

In this paper we consider the problem of routing on an asynchronous bidirectional dynamically changing ring of processors with FIFO queues and with global orientation (i.e., all processors agree on the left and right direction).

Specifically, we assume that the ring has N switches, numbered consecutively $\{0, \ldots, N-1\}$ in clockwise order, with switch i connected to switches numbered $(i-1) \bmod N$ and $(i+1) \bmod N$. There are N processors as well, each of which is associated with exactly one switch. The label of a processor is simply the number of the switch with which it is associated. Messages move between switches: If a switch is open the associated processor is non-active and messages pass through at no cost. If a switch is closed the associated processor is either active or pending and all messages are delivered to the processor for processing. (See Figure 1.) Each processor knows the topology of the network, the value N, and its own label, but must explicitly acquire information about the status of the other processors in the network (i.e., whether they are active, inactive, or pending).

In order to ensure correctness, and more precisely to ensure that messages travel only a bounded number of steps, we assume that there is a unique processor, without loss of generality, the one labeled with value 0, that is always active (see details given later). We use n to denote the number of active and pending processors at any particular time, i.e., n is the effective size of the ring. This model captures the common star-shaped ring Local Area Network (LAN) topology in which all connections pass through a wire center as well as rings consisting of virtual links in optical networks [22].

Below we describe three different DIRSs for the ring network which show a tradeoff between the stretch factor, the adaption cost, and the size of the update messages. The first two algorithms are deterministic, one with adaptation cost zero but worst case stretch factor $\lfloor N/2 \rfloor$, the other with worst case adaptation cost $O(N)$ messages of $O(\log N)$ bits

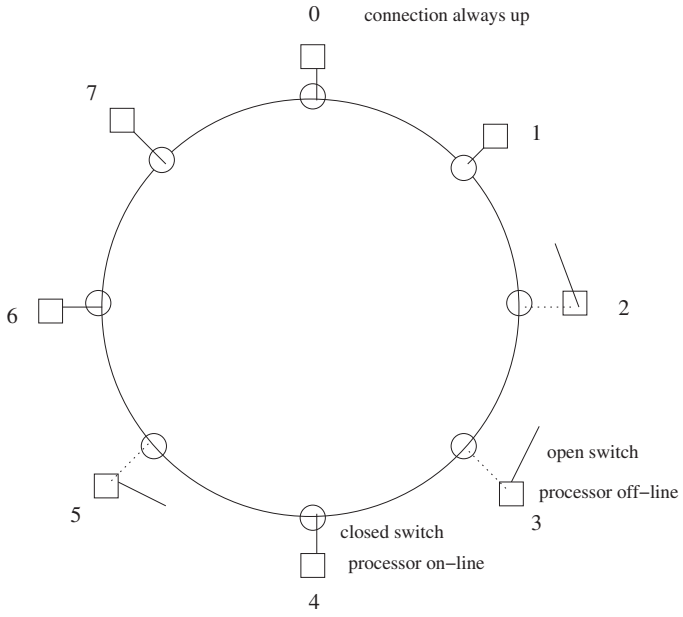

Fig. 1. Switches on and off. Here $N = 8$ and $n = 5$.

but with stretch factor 1. The third algorithm is randomized and has expected amortized adaptation cost $O(k)$ with messages of size $O(k \log N)$ and expected stretch factor $1 + 1/k$, for any integer $k \geq 3$. All of our schemes use only $O(\log N)$ bits per node to store routing table information and require the message headers to contain at most $O(\log N)$ bits.

Regarding the space complexity note that, as in the study of compact routing schemes, the space considered is only that related to the routing scheme, i.e., to the storage of the routing table. In order to speed up routing table look ups, we assume that each table is stored in a cache memory inside the router. We also make the assumption that all of the storage required by the update procedure relies on a different level of storage (either in the router or in the processor itself).

To the best of our knowledge these are the first dynamic routing schemes derived for the above model of the ring. Previous work on dynamic routing has concentrated on other classes of networks such as dynamically growing trees [1] or on general networks [2], [5], [6], [9]. The results on general networks are mostly based upon spanning trees and cluster techniques. When applied to the ring the best of them require polylogarithmic adaptation cost and message size and result in schemes with a polylogarithmic stretch factor.

2. DIRS with Adaptation Cost Zero

In this section we describe a DIRS that has adaptation cost zero (i.e., requires no messages to be sent when processors go on-line or off-line) but has stretch factor $\min\{n-1, \lfloor N/2 \rfloor\}$,

where n is the effective size of the ring. The results of this section are pretty straightforward but are fundamental for understanding the background and the problems that will be faced and solved in the next sections.

We assume the classical node and arc labeling of the IRS for a ring of size N [18], [21]. Every node i is labeled with some unique integer in the set $\{0, \ldots, N-1\}$. Moreover it has two arcs leaving: l_i, the left one in a clockwise orientation of the ring, and r_i, the right one. The (possibly wrapped-around) interval associated to l_i is $[i+1 \bmod N, \lfloor i+N/2 \rfloor \bmod N]$ and to r_i is $[\lfloor i+1+N/2 \rfloor \bmod N, i-1 \bmod N]$. Hence every message destined by i to a node $j \neq i$ in the ring is unambiguously sent either through l_i or through r_i, as j is exactly contained in either the interval associated to l_i or to r_i. As an example consider Figure 1 and the intervals associated to node 0, i.e., $[5, 7]$ to r_i and $[1, 4]$ to l_i. A message from 0 to 2 is therefore sent through arc l_0.

In the DIRS the data message is in the form $M = (D, r, s, x)$ where D is the information to be exchanged, r is the name of the receiver, s is the name of the sender, and x is a value that denotes the number of times M has passed in front of processor 0 (which is always active).

The DIRS consists of an Update and a Routing procedure. The Update procedure (Algorithm 1 in the Appendix) is executed by a non-active processor that wants to go on-line. It consists of getting a fixed label i (line 1), and then of using the classical optimal stretch IRS, i.e., i associates the fixed interval $[i+1 \bmod N, \lfloor i+N/2 \rfloor \bmod N]$ to the left arc (line 2) and the interval $[\lfloor i+1+N/2 \rfloor \bmod N, i-1 \bmod N]$ to the the right arc (line 3) and then becomes active (line 4). Note that no action is required of a processor going off-line. The Routing procedure (Algorithm 2 in the Appendix) is executed by every active processor. When a processor i wants to send a data message to another processor j, it behaves as in the classical optimal stretch IRS, i.e., it checks if the value j is in the interval of the left or right arc and sends it through it (lines 2–7). Moreover, it sets $x := 0$. Note that x is increased by one every time the message passes in front of processor 0 (line 9). An active processor i may also receive a message $M = (D, r, s, x)$ (line 8). If $r = i$, i.e., i is the destination, then M is processed (line 15); otherwise if M comes from the left (right) and $r \in [i+1 \bmod N, s-1 \bmod N]$ (respectively $r \in [s+1 \bmod N, i-1 \bmod N]$), then M is killed as the destination went off-line (line 13); the same holds if $x = 2$, as in this case M has passed in front of processor 0 twice. In all other cases M is forwarded to the opposite direction (lines 10-12).

Theorem 1. *The DIRS defined by Algorithms* 1 *and* 2 *is correct and has the following properties*:

(1) *the space required for the routing tables is at most* $O(\log N)$ *bits per node*;
(2) *the adaptation cost is zero* (*i.e., no update messages are sent*);
(3) *the stretch factor is at most* $\min\{n-1, \lfloor N/2 \rfloor\}$.

Proof. Consider the case in which a receiver j is active during the lifetime of the message M. In this case the DIRS behaves as a normal IRS in a static ring, therefore M is correctly delivered to j. Note also that every processor that goes on-line has a fixed label and the arc labeling is the one of the classical ring IRS. If j is non-active M is

killed by the next active processor in the ring. Note that processor 0 is always active and kills messages that have passed in front of it twice, i.e., with $x = 2$.

The space complexity is $O(\log N)$ bits since every processor stores the value N, its own label, and at most two intervals of $O(\log N)$ bits, one for each out-going arc. No update messages are sent and therefore the adaptation cost is zero. To compute the stretch factor observe that no matter which sequence of changes occurs in the network (processors going on-line or off-line), the longest path a message from i has to travel is to go to processor $j = \lfloor i + N/2 \rfloor \bmod N$. This trivially implies that at any time the stretch factor can be at most $(\lfloor i + N/2 - i \rfloor \bmod N)/1 = \lfloor N/2 \rfloor$ since a message always travels in the same direction and since at most $\lfloor N/2 \rfloor$ processors can be active along that path. Finally observe that $\lfloor N/2 \rfloor$ may be at most $n - 1$, therefore the stretch factor is at most $\min\{n - 1, \lfloor N/2 \rfloor\}$. □

3. Scheme with Linear Adaptation Cost

In this section we describe a DIRS that at quiescence routes with stretch factor 1 and requires $O(\log N)$ bits of space on each processor. For any change to the network (processors going on-line or off-line) between two quiescent states, the worst case adaptation cost is $O(n)$ update messages of $O(\log N)$ bits where n is the maximum number of processors that are active or pending between these two states.

3.1. *The Algorithm*

As we have explained in the previous section, routing can be easily accomplished if every node i sends messages to the left or to the right only, depending on the destination value and on the fixed intervals associated to the arcs leaving i. The loss though is on the stretch factor of the routing path which is $\lfloor N/2 \rfloor$ in the worst case. In this section we introduce a new routing scheme (still based on IRS) which drastically improves on the previous stretch factor by allowing a dynamical update of the intervals of the arcs leaving i. Informally, every such interval will on one side be delimited by the value of the processor which is precisely opposite to i (we call it $op(i)$), i.e., the number of active or pending processors on the left and on the right paths from i to $op(i)$ is equal to within 1. All this is accomplished by ensuring that whenever a change occurs in the network, i.e., when a processor z goes on/off-line, z starts an update phase in which the opposite values of all processors in the ring are dynamically updated.

The algorithm is divided into a Routing procedure and an Update procedure. The Routing procedure (Algorithm 3 in the Appendix) is very similar to Algorithm 2. Informally, the main differences are that the interval associated to every arc leaving a node i now contains a dynamically changing value $op(i)$ (in Algorithm 2 this value was fixed) and that pending processors may only receive and forward messages (in Algorithm 2 a processor coming on-line instantaneously becomes active, i.e., there are no pending processors).

In more detail, an active processor s that wants to send to a processor r a data message $M = (D, r, s, x)$ containing the information D, first sets x to zero and then checks if $s+1 \leq r \leq op(s)$ (line 11) and if this holds then it sends the message to the left (line 12),

else to the right (line 13). (The computations are done mod N.) Every active processor i that receives the data message $M = (D, r, s, x)$ (line 15) either processes it (if $i = r$, line 22) or forwards it (line 19) unless it finds out that the destination processor r went off-line (i.e., if $x = 2$ or if M comes from the left (right) and $r \in [i + 1 \bmod N, s - 1 \bmod N]$ (respectively $r \in [s + 1 \bmod N, i - 1 \bmod N]$); check done in lines 17–18), and in this case it kills the message (line 20). Also in this scheme processor 0 increases the value x (line 16). Every pending processor i either forwards the data message (line 5) or kills it (line 6) if the destination went off-line or if $r = i$, i.e., i is the receiver (this check is done in lines 3–4). As an example consider Figure 1. Here $n = 5$ and the intervals associated to node 0 are $[5, 7]$ to r_0 and $[1, 4]$ to l_0, (i.e., $op(0) = 4$), as there are two processors on-line both on the left path from 0 to 4 and on the right path from 0 to 5. A message from 0 to 2 is thus sent through arc l_0.

The update strategy (Algorithm 4 in the Appendix) is more complicated and is divided into three phases. We discuss the case where a processor goes on-line in detail. The case where a processor goes off-line is handled similarly. Recall that a pending processor cannot go off-line, i.e., it must first complete its update procedure.

Informally, whenever a processor wants to go on-line it becomes pending and gets its (fixed) label. For simplicity, let us first assume that a single processor, e.g., i goes on-line. Trivially, in this case the new (exact) opposite value that will have to be stored by each processor will either remain unchanged or will become the old opposite value of its right or left neighbor. All this will depend on the new oddness or evenness of the number n of processors in the ring and on the interval i belongs to. To obtain such an update, processor i has to start sending messages so that every processor may collect the values of its right and left neighbors together with their opposite values. The last thing to observe is that when more than one processor goes on-line the updates have to be sequentialized. This is realized by dividing the update into different phases and by letting only messages related to the update started by the processor with the maximal label and in the highest phase go through and temporarily stopping and buffering all other messages.

In more detail, every processor i that goes on-line becomes pending and gets its fixed label i (line 2). It then sends a Phase 1 message (with values initialized to -1, line 4) in the direction of the orientation (e.g., to the left). This message collects the value of the left and right neighbors of i (called l^i and r^i, respectively), their opposite values ($op(l^i)$ and $op(r^i)$, respectively), and their actual knowledge on the oddness or evenness of the ring ($even(l_i)$ and $even(r_i)$, where $even(x) = 1$ if x knows n is even, 0 otherwise). If i gets the message back (line 28) it moves to the next phase since it has won a "race" and it is the only processor going to the next phase. The mechanism used to win the race is simple: if a processor in Phase 1 receives an updated message started by some other processor it lets it through only if the sender's label is bigger (lines 7–9 and 11) or if the sender is in a higher phase (lines 13–26). Every processor in a higher phase stops and buffers in a FIFO queue Phase 1 messages (lines 55 and 72). Therefore a unique processor (the one with the maximal label among pending processors) may move to Phase 2 (and consequently to Phase 3).

We assume i is the unique processor that receives back its Phase 1 message. If necessary (i.e., only if some of the values have not been set yet, see lines 30–39), it updates the values of the variables it has just collected, i.e., l_i, r_i, $op(l_i)$, $op(r_i)$, $even(l_i)$

and $even(r_i)$. It then computes its new opposite value $op(i)$. This is done, without loss of generality, by considering $even(r_i)$ (lines 40–41) and by setting $op(i)$ to $op(r_i)$ if $even(r_i) = 1$, i.e., if the old ring had an even number of processors or vice versa to $op(l_i)$. The variable $even(i)$ is also set to the value opposite to $even(r_i)$ as the new evenness of the ring has to be changed and updated (lines 40–46).

At the very end of the algorithm, every processor j has to contain consistent and exact information on j, $op(j)$, $even(j)$, l_j, $op(l_j)$, $even(l_j)$, r_j, $op(r_j)$, and $even(r_j)$. Therefore, the aim of Phases 2 and 3 is now to propagate the update from i to the remaining processors. Moreover, to be consistent, i will also have to update its right and left opposite values ($op(r_i)$ and $op(l_i)$), as they may change during these new phases.

More precisely, the Phase 2 message first has the aim of updating the opposite value of each processor, which may either remain the same or become the opposite value of the right or left neighbor depending on the new oddness or evenness of the ring (lines 110–118 and 123–131). Moreover both Phase 2 and 3 messages are used (one for each direction) for the update on every processor of the left and right values, their related opposite and evenness values.

All this is done by sending messages containing the name of the local processor i, its opposite $op(i)$ and evenness $even(i)$ values (lines 47–53 and 66–70), and by dynamically updating it. In other words, if a message is sent to the right, j, the next processor receiving it, will set its local variables r_j to i, $op(r_j)$ to $op(i)$, and $even(r_j)$ to $even(i)$ (lines 105–109 and 136–140), and will forward a message containing j, $op(j)$, and $even(j)$ (lines 133 and 150).

When i completes Phase 3 it becomes active, updates all the buffered messages, and lets them through. In this way the blocked updates may proceed (lines 87–89). This mechanism essentially sequentializes all insertions.

3.2. *Algorithm Analysis*

Theorem 2. *Let $\mathcal{D}$ be the DIRS defined by Algorithms* 3 *and* 4, *together with the appropriate procedure for going off-line. Then $\mathcal{D}$ is correct and has the following properties*:

(1) *the stretch factor is* 1;
(2) *the space required for the routing tables is at most $O(\log N)$ bits per node*;
(3) *the worst case adaption cost of any change to the network between two quiescent states is $O(n)$ messages of $O(\log N)$ bits each, where n is the maximum number of processors that are active or pending between these two states*.

Proof. The general correctness of the Routing procedure derives from the fact that the DIRSs are based on the classical IRSs. If the receiver r is not on-line, then there exists a pending or an active processor (the one immediately after r's position) that by looking at the side the message comes from, and at the sender's label, will realize r went off-line. In any case processor 0 is always active. On the other hand, if r is on-line it will eventually receive the message and in the case where it is pending it will kill it. Therefore all data messages are delivered correctly. Since no pending processor can go off-line without completing its update procedure, update messages are all eventually delivered.

To show that the Update procedure is correct it is sufficient to show that (i) at most one processor enters Phase 2 at a time, (ii) all pending processors eventually enter Phase 2

(and thereafter Phase 3 and complete their update), and (iii) at the completion of Phase 3 of an update the routing tables of all active processors are correct (ignoring pending processors).

Proof of (i). We assume by contradiction that at least two processors x and y enter Phase 2. If this is the case it means that both have completed Phase 1, therefore they both got their values back. Without loss of generality, we assume $x < y$. If this is the case, then the message from y could have passed in front of x but the message from x could not (since it found a processor in the same phase but with a bigger value therefore it had to stop), therefore it passed by before y got up. If this is the case, then x got into Phase 2 before the message for y passed by, and therefore this message must have stopped and must have been stored in x's FIFO queue, therefore yielding a contradiction. Symmetrically if $y < x$ only one processor wins. Moreover, only processors in Phase 2 can move to Phase 3. This can be generalized to many pending processors therefore messages either stop at a processor in Phase 2 or 3 or at others in Phase 1 that have bigger values. When the processor has completed the update it removes buffered messages (that can only be of processors in Phase 1). A new "race" can then start. Also observe that if at least one other processor sent a Phase 1 message then i contains at least one other processor's Phase 1 message in its buffer, i.e., new updates can start.

Proof of (ii). A pending processor eventually sends a Phase 1 message. All Phase 1 messages travel around the ring in a single direction through FIFO queues. A Phase 1 message can be blocked by either a processor in Phase 2 or 3 or by another Phase 1 processor with a higher identity. In the first case the message is unblocked and makes progress as soon as the blocking processor completes its Phase 3. In the second case the Phase 1 blocking processor eventually enters Phase 2 and then 3, and the message makes progress. If not, it must be the case that a cycle of messages blocked by a processor's Phase 1 message exists but this cannot occur since the message from the Phase 1 processor with the highest identity always makes progress.

Proof of (iii). We now show that the routing tables of active processors are correct at the completion of an update. Assume that a processor i wakes up and sends a Phase 1 message M around. If it is the only pending or largest identity processor it will receive back the correct values of its right and left neighbor since no other processor is in Phase 2 or 3. On the other hand, let us assume that M gets to another pending processor j with a bigger value. M will stop at j and i will eventually get it after j's update, but the correct values of its neighbors might have changed in the meantime. On the other hand, every time a processor starts Phase 2 and 3 it updates the values of any active or pending processors x in the ring, i.e., the value $op(x)$ of the opposite processor, the values r^x, l^x of the right and left active neighbors and $op(r^x)$, $op(l^x)$ their opposite values, respectively, and the evenness values $even(x)$, $even(r^x)$, and $even(l^x)$. Therefore i will have its values updated. Note that Phases 2 and 3 have to be sequentialized as in Phase 2 every processor updates its opposite value and therefore a Phase 3 message passing by earlier would collect a wrong value. The same holds if other updates start before i gets its Phase 1 message back, and the temporary variables will be replaced if necessary. Therefore at the end i will be able to choose between old and new values if there are any and therefore it will be able to take into account all the updates that took place in the meantime.

We are now ready to show points (1)–(3) in the statement of the theorem, namely: (1) the stretch factor is 1; (2) the space required for the routing tables is at most $O(\log N)$

bits; and (3) the amortized adaptation cost per update is $O(n)$ messages of $O(\log N)$ bits.

Proof of (1). This follows immediately from point (iii) above. If at the end of each update the opposite values are correct and the system reaches quiescence, then the stretch factor is obviously 1.

Proof of (2). The space complexity is straightforward since every processor stores a constant number of values of at most $O(\log N)$ bits each.

Proof of (3). Every processor that goes on/off-line generates at most $O(n)$ messages since the update consists of three phases in each of which the size of the ring is at most n (as this update occurs during two quiescent states). Messages are $O(\log N)$ bits each. Therefore the worst case adaption cost of any change to the network is $O(n)$ messages of $O(\log N)$ bits each. □

Finally note that if the system runs synchronously the worst case time complexity of the algorithm, computed between two quiescent states, is $O(mn)$ steps, where m is the number of changes occurring between these two states and n is the maximum number of active or pending processors. The worst case occurs when all m updates are sequentialized as in this case each single update requires $O(n)$ steps.

4. Scheme with Constant Expected Adaptation Cost

In this section we describe a randomized DIRS that at quiescence routes with expected stretch factor $1 + 1/k$ and requires an expected amortized $O(k)$ messages containing $O(k \log N)$ bits each for each change in the ring, for any integer $k \geq 3$. If k is chosen to be constant the expected adaptation cost is then constant with update messages of size $O(\log N)$. Achieving a smaller expected stretch factor is possible but this requires more messages of larger size.

4.1. *The Algorithm*

The routing scheme we use is again based upon the classical optimal stretch IRS for the ring. Every node i assigns to the left arc the interval $[(i + 1) \bmod N, op(i)]$ and to the right arc $[op(i) + 1 \bmod N, (i - 1) \bmod N]$, where $op(i)$ is an estimate of i's true opposite value accurate to within a factor of approximately $1/k$. This value is updated with probability proportional to an estimate of the number of active processors in the ring by sending a constant number of messages of size $O(k \log N)$ all the way around the ring. The probability is chosen so that the expected adaptation cost is $O(k)$ and the expected stretch factor is less than $1 + 1/k$. We discuss the case where a processor goes on-line in detail. The case where a processor goes off-line is analogous and is only sketched below. Recall that a pending processor cannot go off-line, i.e., it must first complete its update procedure.

The DIRS consists of two different algorithms: one used to route messages and one used for the update. The Routing procedure is the same as for the linear adaptation cost algorithm (see also Algorithm 3 in the Appendix) with the only difference that in this case $op(i)$ is an estimate of i's true opposite value.

The Update procedure (Algorithm 5 in the Appendix) is more complicated. Informally, the general idea is that at every instant processors store opposite values which may not be perfectly accurate. Exact opposite values are recomputed only after a certain number of network changes, i.e., pending processors flip coins to decide whether or not to start a new update. This obviously implies that that the stretch factor may not always be 1, but still is very small, and that the amortized cost of the updates is not too high. The update strategy is again divided into three phases, and all updates are sequentialized by using a mechanism similar to the one presented for Algorithm 4. In Phase 1 a processor computes the number of on-line processors, in Phase 2 it collects a subset of their labels (roughly equally spaced among the on-line processors), and finally in Phase 3 it sends these values around the ring so that every processor can compute from them its new opposite value.

More formally, every pending processor i has to do the following things (see Algorithm 5 in the Appendix): it sends a message $R = (1, i)$ to the next active processor on the left. The first active processor j receiving R, knows from the value 1 it is a request for its n and opposite values (lines 4–5 and 102). Processor j replies sending to the right an answer message $A = (2, i, j, n, op(j))$ that contains the requested values n and $op(j)$ and the label 2 to state it is an answering message (lines 102–103). Processor i waits for this message, and stores the received values (lines 33–34). It then flips a coin with probability of heads equal to $\min\{1, 10k/n\}$ (line 37). If it gets a tail it replies to all buffered messages and it becomes active (lines 97–99). Otherwise, it starts an update similar to that of Algorithm 4 by sending a Phase 1 message $U_1 = (3, n_0, n_1, i)$ that contains the value 3 to state is a Phase 1 message, a counter n_0 for the number of active processors in Phase 1, n_1 for the ones pending during Phase 1 (both initialized to 0), and its name i (lines 39–40). The counters are increased by the receiving processors. Note that update messages are always forwarded in the same direction they come from, as in Algorithm 4.

Consider what happens when a single processor starts an update. In this case it gets back the Phase 1 message U_1 containing the number of active processors, n_0, and pending processors, n_1, i.e., i's new estimate on n is now $n_0 + n_1$, it then sends a Phase 2 message $U_2 = (4, \lfloor (n_0 + n_1)/10k \rfloor, d, V, i)$ that contains a value 4 to state it is a Phase 2 message, a value $\lfloor (n_0 + n_1)/10k \rfloor$ that defines the intervals of values to take, a counter d (initialized to 0), a vector V of processors labels (initially containing i), and its value i (line 70). Message U_2 (line 70) globally collects $\lceil (n_0 + n_1)/\lfloor (n_0 + n_1)/10k \rfloor \rceil$ processor labels of nodes that are active or pending during Phase 1. This is obtained by storing in U_2 the value $\lfloor (n_0 + n_1)/10k \rfloor$, and by adding a counter d, that is increased only by processors that are on-line during Phase 1 (lines 54–61 and 118–123). When $d = \lfloor (n_0 + n_1)/10k \rfloor - 1$ the receiving processor adds its label to U_2 and sets $d := 0$ (lines 55–56 and 119–120). Finally during Phase 3, i sends an update message $U_3 = (5, n, V, i)$ that contains a value 5 indicating it is a Phase 3 message, its n value, the vector V containing the labels it gathered in Phase 2, and its label i (line 82). Whenever an active or pending processor j receives a U_3 message, it updates its $op(j)$ and n values (lines 21–26, 62–67, and 124–129). A pending processor awaiting an answer A message (i.e., values $op(j)$ and n of its left active neighbor) that receives a Phase 1 message (lines 7–10) is counted in n_1 and gets activated at the end of Phase 3 without flipping a coin (i.e., its values will be updated by some other processor going on-line). A pending processor awaiting an

answer A message that receives a Phase 2 (lines 17–20) or Phase 3 (lines 21–26) message waits until the completion of the update at which point it will have collected the new n and opposite value and it will flip a coin (lines 36–37) and eventually start a new update if the result is heads (39–99). Whenever it receives its A message it kills it (lines 42–43).

The case in which more than one processor sends a Phase 1 message is solved using an ordering of the requests based on the largest label value as in Algorithm 4. In this case the update performed by the "winning" processor acts as an update for all the other processors that entered Phase 1 and it is not necessary for them to continue to Phases 2 and 3 (lines 48–53).

The procedure for a processor i going off-line is analogous to the above. Processor i flips a coin with probability $\min\{1, 10k/n\}$ of heads (where n is the most recent estimate of the size of the ring). If it gets a tail it goes off-line. Otherwise, it begins an update procedure as before. Phase 1 counts the active and pending processors. Assuming i moves to Phase 2, processor labels are collected and in Phase 3, the opposite values of active and pending (during Phase 1) processors are updated. At the completion of Phase 3, i goes off-line. Processors deciding to go off-line during an update wait until the completion of the update to flip their coin. If more than one processor enters Phase 1, the one with the largest label proceeds to Phase 2 and all such pending processors are updated together. Note that pending processors that are going off-line add -1 during the counting phase (i.e., during the computation of $n_0 + n_1$).

4.2. *Algorithm Analysis*

Theorem 3. *For any integer $k \geq 3$, let $\mathcal{R}$ be the DIRS defined by Algorithm* 3 *and* 5, *together with the appropriate procedure for going off-line. Then $\mathcal{R}$ is correct and has the following properties*:

(1) *the expected stretch factor is at most* $1 + 1/k$;
(2) *the space required for the routing tables is at most* $O(\log N)$ *bits per node*;
(3) *the expected amortized number of messages sent per update is* $O(k)$ *of* $O(k \log N)$ *bits*.

Proof. The proof of correctness of the Routing procedure is the same as that for Algorithm 3 given above.

We now show that all pending processors eventually go on/off-line depending upon the action they request. A processor wishing to become active sends a request R message. Four possible situations may arise: (a) It receives back an answer A to its R message and flips a coin with outcome tails. (Note that since processor 0 is always active, the R message is always received by some active processor.) In this case it immediately becomes active. (b) It receives back an answer A to its R message and flips a coin with outcome heads. In this case it enters Phase 1. By an argument similar to that given for Algorithm 4 at most one processor enters Phase 2 and its update runs to completion. At the completion of the update, all processors that were in Phase 1 at the beginning of the update will become active. (c) It does not receive an answer A message but it receives a Phase 1 message. In this case it participates in the update and upon its completion becomes active. (d) It does not receive an answer A message but it receives a Phase 2 or 3 message. In this case, at the completion of the update, it flips a coin and depending on

the outcome ends up in case (a) or (b) above. A processor wishing to go off-line waits until an update in progress is completed if one is in progress and then flips its coin. At this point two situations can arise: (a) The coin flip outcome is tails in which case it goes off-line. (b) The coin flip outcome is heads in which case it enters Phase 1. As before a single processor will eventually enter Phase 2 and upon completion inform processors that they may go off-line.

Proof of (1). We now show that the routing tables of active processors are correct to within an expected stretch factor of $1 + 1/k$. Consider the system at a quiescent state and assume that the last update was done by processor i, i.e., i was the (unique) last processor to enter into Phase 2 of the update procedure. Let n_0 be the number of active processors counted by i during its Phase 1 and let n_1 be the number of pending processors which are going on-line minus the number of pending processors that are going off-line counted by i during its Phase 1. (Note that active processors that wish to go off-line but have already received i's Phase 1 or 2 messages are still active until after the update is completed and are counted in n_0.) Let n_2 be the change in the size in the ring since the end of Phase 1 for i, i.e., the number of processors that became active minus the number that went off-line between the time that i's Phase 1 message passes by the processor and the quiescent state we are examining. Note that all of these processors flip a coin with probability of heads $\min\{1, 10k/(n_0 + n_1)\}$. The result of all of these coin flips is tails. Otherwise, at least one of these processors would have initiated an update, i.e., a contradiction. Therefore the absolute value of expected value of n_2 is less than or equal to $(n_0 + n_1)/10k$.

Letting $n_0+n_1 = \nu$, $D = \lfloor \nu/(10k) \rfloor$, we note that there are at most $\lambda = \lceil \nu/D \rceil$ labels in V, which we denote by $v_0, \ldots, v_{\lambda-1}$. To calculate $op(x)$, we let v_j, $j \in \{0, \ldots, \lambda-1\}$, be the first processor after x in clockwise order around the ring that belongs to V, if $x \notin V$ (if $x \in V$ we let $v_j = x$), and finally set $op(x) = v_{(j+\lfloor \lambda/2 \rfloor) \bmod \lambda}$. We now calculate the minimum distance between x and $op(x)$. The distance between x and $op(x)$ can be considered to be made up of $\lfloor \lambda/2 \rfloor + 1$ distances, the first from x to v_j, which is between 0 and $D-1$ (0 if $x = v_j$) and the remainder being of length exactly D except that one may be as small as 1 (if D does not divide ν evenly). Thus, the minimum distance is given by

$$1 + (\lfloor \lambda/2 \rfloor - 1)D \geq 1 + (\lambda/2 - 3/2)D \geq 1 + (\nu/(2D) - 3/2)D.$$

In the worst case the longer distance between $op(x)$ and x stays the same (respectively is increased by n_2), but the shorter distance decreases by n_2, whose expected value is $\nu/(10k)$ (respectively remains the same). Thus, the expected stretch is bounded by

$$\frac{\nu - (1 + (\nu/(2D) - 3/2)D}{1 + (\nu/(2D) - 3/2)D - \nu/(10k)} \leq \frac{\nu/2 + 3D/2}{\nu/2 - 3D/2 - \nu/(10k)} \leq \frac{10k+3}{10k-5}.$$

It is easy to verify that this is bounded by $1 + 1/k$ for any integer $k \geq 3$ (respectively with similar computations we obtain that the expected stretch is bounded by the same value).

Proof of (2). The space complexity is straightforward since every processor stores a constant number of values of at most $O(\log N)$ bits each.

Proof of (3). We now prove that the expected amortized number of messages sent per update is $O(k)$. Messages are of size $O(k \log N)$ since R, A, and Phase 1 messages (U_1) have at most $3 + 4 \log N$ bits. Phase 2 and 3 messages (U_2 and U_3) have at most $3 + (20k + 3) \log N$ bits as the number of labels collected in V is at most $20k$. As a matter of fact, for $n_0 + n_1 \geq 10k$, and $n_0 + n_1 = q10k + r$, with $r < 10k$, r and q integers, we have:

$$\left\lceil \frac{n_0 + n_1}{\lfloor (n_0 + n_1)/10k \rfloor} \right\rceil = \left\lceil \frac{q10k + r}{\lfloor (q10k + r)/10k \rfloor} \right\rceil = \left\lceil \frac{q10k + r}{\lfloor q + r/10k \rfloor} \right\rceil = \left\lceil 10k + \frac{r}{q} \right\rceil$$
$$< \left\lceil 10k + \frac{10k}{q} \right\rceil = \left\lceil 10k \left(1 + \frac{1}{q}\right) \right\rceil \leq \lceil 20k \rceil = 20k.$$

The expected number of messages sent per update can be bounded as follows. A pending processor is responsible for sending at most one R message and at most one A message. Moreover, it sends at most two Phase 1 messages, i.e., its own if it does flip a coin or that of some pending processor behind it if it does not, plus the Phase 1 message of the eventual "winner" of the Phase 1 "race," and at most one Phase 2 and 3 messages. After flipping its coin, with the probability of heads equal to $\min\{1, 10k/n\}$, where n is the processors estimate for the size of the ring determined during the previous successful update, it generates an update that may proceed through all three phases. Let n' be the number of changes that have occurred in the ring since the value n was determined including changes occurring up to the point where the processor receives back its "winning" Phase 1 message. The successful update is responsible for a total of $3(n + n')$ messages for each of the three phases. During this period $1 + n'$ processors go on-line or off-line. Note that processors arriving during Phase 2 or 3 are responsible for their own messages before they flip their coin.

Therefore the expected amortized cost per update is at most

$$6 + \min\left\{1, \frac{10k}{n}\right\} \frac{3(n + n')}{1 + n'} = O(k).$$

For the space complexity observe that every processor stores its label, the estimated value of n, and an opposite value all of $O(\log N)$ bits plus some extra variables of $O(1)$ bits. At run time it needs another $O(\log N)$ bits for local computation. □

5. Conclusions

In this paper we have considered the problem of routing in an asynchronous dynamically changing ring of processors. We introduced a new technique, Dynamic Interval Routing, and applied it to the ring. We presented three algorithms for rings of maximum size N: the first two are deterministic, one with adaptation cost zero but worst case stretch factor $\lfloor N/2 \rfloor$, the other with worst case adaptation cost $O(N)$ messages of size $O(\log N)$ bits and stretch factor 1. The third is a randomized algorithm that uses update messages of size $O(k \log N)$, has adaptation cost $O(k)$, and expected stretch factor $1 + 1/k$. All schemes require $O(\log N)$ bits per node for storing the routing information and all messages have headers of size $O(\log N)$ bits.

Observe that the techniques introduced can be easily extended to the case of the ring of rings networks. It remains an open problem to study whether the tradeoffs established by our randomized algorithm hold in the deterministic setting and to see to which other topologies the above techniques can be applied. Also, it would be interesting to find tight lower bounds for the problem.

Appendix

Algorithm 1 /* Update procedure for a processor i going on-line*/

1 get fixed label i;
2 associate interval $[(i+1) \bmod N, \lfloor i + N/2 \rfloor \bmod N]$ to the left arc;
3 associate interval $[\lfloor i + 1 + N/2 \rfloor \bmod N, (i-1) \bmod N]$ to the right arc;
4 become active;

Algorithm 2 /* Routing procedure for active processor i*/

```
1 repeat
2       if willing to send a data message M to a node r then
3         if (i + 1) mod N ≤ r ≤ ⌊i + N/2⌋ mod N
4           then send M = (D, r, i, 0) to the left
5           else to the right
6         fi
7       fi;
8       if receiving M = (D, r, s, x) from direction then
9         if r ≠ i then if i = 0 then x := x + 1;
10                      if x ≠ 2 and ((direction = left) and (r ∉ [(i + 1) mod N, (s − 1)mod
                        N]))
11                        or ((direction = right) and (r ∉ [(s + 1) mod N, (i − 1) mod N]))
12                                then forward M = (D, r, s, x) to opposite (direction)
13                                else kill M /* r went off-line */
14                      fi
15                  else process M
16        fi
17      fi
18 until off-line;
```

Algorithm 3 /* Routing procedure for processor i */

```
1 if pending
2   then if  receiving a data message M = (D, r, s, x) from direction
3            then if [((direction = left) and (r ∉ [(i + 1) mod N, (s − 1) mod N])) or
4                 ((direction = right) and (r ∉ [(s + 1) mod N, (i − 1) mod N]))] and (r ≠ i)
5                                  then forward M = (D, r, s, x) to opposite (direction)
6                                  else kill M /* r went off-line or M is an old message */
7                 fi
8        fi
9   else  /* i is active */
10        if sending a data message M = (D, r, i, 0) to a processor r
11          then if (i + 1) mod N ≤ r ≤ op(i)
12                 then send M = (D, r, i, 0) to the left
```

```
13                  else to the right
14              fi
15          else if receiving M = (D, r, s, x) from direction
16                  then if r ≠ i then if i = 0 then x := x + 1;
17                          if x ≠ 2 and ((direction = left) and (r ∉ [(i + 1) mod N, (s − 1)mod
                            N]))
18                            or ((direction = right) and (r ∉ [(s + 1) mod N, (i − 1) mod N]))
19                                  then forward M = (D, r, s, x) to opposite (direction)
20                                  else kill M /* r went off-line */
21                            fi
22                          else process M
23                      fi
24              fi
25      fi
26 fi;
```

Algorithm 4 /* Update procedure for processor i */

```
/*  Phase 1 messages are in the form M = (1, r, a, op(a), even(a), b, op(b), even(b)), where r
    is the receiver, and at the end a and b will be r's left and right active processors; even(x) = 1
    if x is presently assuming the ring has an even number of processors; Phase 2 messages are
    in the form M = (2, r, op(r), prev(r), k, op(k), even(k), oldop(k)), where r is the receiver,
    prev(r) is the previous active neighbor with respect to the direction in which r sends M, k
    is a processor, and oldop(k) its old opposite value; finally Phase 3 messages are in the form
    M = (3, r, k, op(k), even(k)). */
1 if pending then /* the new pending processor has to get new values */
2   get value i;
3   r^i, l^i, op(i), op(l^i), op(r^i), even(i), even(r^i), even(l^i), oldop(i) := −1;
4   send M = (1, i, −1, −1, −1, −1, −1, −1) to the left;
5   /* sending a new update Phase 1 message */
6   repeat
7           if receiving M = (1, r, a, op(a), even(a), b, op(b), even(b))
8             /* another processor r ≠ i is in Phase 1 */
9             then if i > r
10                        then buffer M
11                        else forward M
12                  fi
13            else if receiving M = (2, r, op(r), prev(r), k, op(k), even(k), oldop(k)) or
14                    M = (3, r, k, op(k), even(k)) /* i receives a Phase 2 or 3 message */
15                    then /* i updates left or right values since some other
16                            processor has gone on-line before it */
17                         if receiving M from the left
18                           then l^i := k;
19                                op(l^i) := op(k);
20                                even(l^i) := even(k)
21                           else r^i := k;
22                                op(r^i) := op(k);
23                                even(r^i) := even(k)
24                         fi;
25                         forward M
```

```
26                fi
27          fi
28  until receiving M = (1, i, a, op(a), even(a), b, op(b), even(b));
29  /* i got back its Phase 1 message */
30  if l^i = −1 then /* i's left values have not been updated yet */
31                l^i := a;
32                op(l^i) := op(a);
33                even(l^i) := even(a)
34  fi;
35  if r^i = −1 then /* i's right values have not been updated yet */
36                r^i := b;
37                op(r^i) := op(b);
38                even(r^i) := even(b)
39  fi;
40  /* asking the right neighbor for the oddness or evenness */
41  if even(r^i) = 1 /* i gets values of right neighbor */
42    then op(i) := op(r^i);
43          even(i) := ¬even(r^i)
44    else op(i) := op(l^i);
45          even(i) := ¬even(l^i)
46  fi;
47  /* a Phase 2 message has to be sent */
48  if even(i) = 1
49    then prev(i) := l^i;
50          send M = (2, i, op(i), prev(i), i, op(i), even(i), oldop(i)) to the right
51    else prev(i) := r^i;
52          send M = (2, i, op(i), prev(i), i, op(i), even(i), oldop(i)) to the left
53  fi;
54  repeat
55          if receiving M with r ≠ i then buffer M fi
56  until receiving M = (2, i, op(i), prev(i), k, op(k), even(k), oldop(k));
57  /* k is the previous active processor */
58  if receiving M from the right
59    then /* update right values */
60          op(r^i) := op(k);
61          even(r^i) := even(k)
62    else /* update left values */
63          op(l^i) := op(k);
64          even(l^i) := even(k)
65  fi;
66  M := (3, i, i, op(i), even(i));
67  if even(i) = 1
68    then send M to the left
69    else to the right
70  fi;
71  repeat
72          if receiving M with r ≠ i then buffer M fi
73  until receiving M = (3, i, k, op(k), even(k));
74  if receiving M from the right
75    then /* update right values */
```

```
76          op(r^i) := op(k);
77          even(r^i) := even(k)
78    else /* update left values */
79          op(l^i) := op(k);
80          even(l^i) := even(k)
81  fi;
82  if l^i = r^i then /* there is only another active processor, i.e., 0 */
83                    oldop(i) := op(i);
84                    op(i) := l^i;
85                    op(l^i) := op(r^i) := i
86  fi;
87  process buffered messages in FIFO order and eventually update their old values;
88  let messages eventually go through;
89  become active
90  else /* i is active */
91  repeat
92       if receiving M = (1, r, a, op(a), even(a), b, op(b), even(b)) /* Phase 1 message */
93          then if r = i then kill M /* r cannot go off-line */
94                        else /* M has to be updated */
95                             if a = op(a) = even(a) = −1
96                               then M := (1, r, i, op(i), even(i), i, op(i), even(i))
97                                    /* i is r's left processor */
98                               else M := (1, r, a, op(a), even(a), i, op(i), even(i))
99                                    /*i is r's possible right processor */
100                            fi;
101                            forward M
102               fi
103       else if receiving M = (2, r, op(r), prev(r), k, op(k), even(k), oldop(k))
104               /* Phase 2 message */
105               then if coming from the right
106                       then /* update right and local values */
107                            r^i := k;
108                            op(r^i) := op(k);
109                            even(i), even(r^i) := even(k);
110                            if k = op(r) /* the previous active processor is op(r) */
111                              then oldop(i) := op(i);
112                                   op(i) := r
113                              else if i ∈ (op(r), prev(r)]
114                                        /* i is between op(r) and r */
115                                        then oldop(i) := op(i);
116                                             op(i) := oldop(k)
117                                   fi
118                            fi
119                       else /* update left and opposite values */
120                            l^i := k;
121                            op(l^i) := op(k);
122                            even(i), even(l^i) := even(k);
123                            if i = op(r) /* i is opposite to r */
124                              then oldop(i) := op(i);
125                                   op(i) := r
```

```
126                          else if i ∈ [prev(r), op(r))
127                                  /* i is between r and op(r) */
128                                  then oldop(i) := op(i);
129                                       op(i) := oldop(k)
130                              fi
131                      fi
132              fi;
133              forward M = (2, r, op(r), prev(r), i, op(i), even(i), oldop(i))
134          else if receiving M = (3, r, k, op(k), even(k))
135              /* Phase 3 message */
136              then if coming from the right
137                      then /* update right values */
138                           r^i := k;
139                           op(r^i) := op(k);
140                           even(r^i) := even(k)
141                      else /* update left values */
142                           l^i := k;
143                           op(l^i) := op(k);
144                           even(l^i) := even(k)
145                      fi;
146                      if r^i = l^i then oldop(i) := op(i);
147                                        op(i) := r^i;
148                                        op(r^i) := op(l^i) := i
149                      fi;
150                      forward M = (3, r, i, op(i), even(i))
151          fi
152      fi
153  fi
154 until off-line
155 fi;
```

Algorithm 5 /* Update procedure for processor i */

/* Every message contains as its first value a message label. Request messages are $R = (1, r)$, where r is the receiver's (and in this case also the sender's) name; answer messages are $A = (2, r, s, n, op(s))$, where s is the sender's name and $op(s)$ is its opposite value; Phase 1 messages are in the form $U_1 = (3, n_0, n_1, r)$, where n_0 (n_1) is the number of active (pending) processors during Phase 1; Phase 2 messages are in the form $U_2 = (4, \lfloor (n_0 + n_1)/10k \rfloor, d, V, r)$, where d is a counter and V a vector of processors labels; finally Phase 3 messages are in the form $U_3 = (5, n, V, r)$. */

```
1 if pending then
2   get value i;
3   a, exit := 0;
4   send R = (1, i) to the left;
5   /* this is a request of the n and op(next(i)) values */
6   repeat
7       if receiving a message U_1 = (3, n_0, n_1, r)
8         then a := 1 /* message A for i will have to be ignored and the update will be
            global */ ;
9              forward U_1 = (3, n_0, n_1 + 1, r)
```

```
10        fi;
11        if receiving a message U_2 = (4, ⌊(n_0 + n_1)/10k⌋, d, V, r)
12          then if a = 1 /* i may have to be added to V as it is pending during Phase 1 */
13                  then if ⌊(n_0 + n_1)/10k⌋ = d − 1
14                          then forward U_2 = (4, ⌊(n_0 + n_1)/10k⌋, 0, V ∪ {i}, r)
15                          else  forward U_2 = (4, ⌊(n_0 + n_1)/10k⌋, d + 1, V, r)
16                       fi
17                  else /* a = 0, i.e., i was not pending during Phase 1 */
18                       forward U_2 = (4, ⌊(n_0 + n_1)/10k⌋, d, V, r)
19               fi
20        fi;
21        if receiving a message U_3 = (5, n, V, r)
22          then get n;
23               get op(i) from V;
24               exit := 1;
25               forward U_3
26        fi;
27        if receiving R = (1, r) then buffer R fi;
28        if receiving A = (2, r, s, n, op(s)) and r ≠ i
29          then if r is between i and s then kill A
30                                       else buffer A
31        fi;
32        if (a = 1) and receiving A = (2, i, s, n, op(s)) then kill A fi
33  until exit or ((a = 0) and receiving A = (2, i, s, n, op(s)));
34  if not exit then get new n and op(i) value from A;
35  if ((a = 1) and exit) then goto 2 /* i was pending during Phase 1 */
36  else /* i was pending during Phase 2 or 3 or received A */
37  1: flip the coin;
38  a, exit := 0;
39  if head /* obtained with probability min{1, 10k/n} */
40    then send an update message U_1 = (3, 0, 0, i)
41         repeat
42              if receiving A = (2, r, s, n, op(s))
43                then if r = i or r is between i and s then kill A /* old message or r is off-
                       line */
44                                                    else buffer A
45                     fi
46              fi;
47              if receiving R = (1, r) then buffer R fi;
48              if receiving U_1 = (3, n_0, n_1, r) and r ≠ i
49                then if r < i then kill U_1
50                                else forward U_1 = (3, n_0, n_1 + 1, r);
51                                     a := 1
52                     fi
53              fi;
54              if receiving U_2 = (4, ⌊(n_0 + n_1)/10k⌋, d, V, r)
55                then if a = 1 then if ⌊(n_0 + n_1)/10k⌋ = d − 1
56                                   then forward U_2 = (4, ⌊(n_0 + n_1)/10k⌋, 0, V ∪ {i},
                                     r)
57                                   else forward U_2 = (4, ⌊(n_0 + n_1)/10k⌋, d + 1, V, r)
```

```
58                                  fi
59                              else forward U2 = (4, ⌊(n0 + n1)/10k⌋, d, V, r) /* a = 0 */
60                          fi
61                  fi;
62                  if receiving U3 = (5, n, V, r)
63                    then get n;
64                          get op(i) from V;
65                          exit := 1;
66                          forward U3
67                  fi
68          until receiving U1 = (3, n0, n1, i) or exit;
69          if exit = 1 and a = 1 then goto 2 /* i was pending during Phase 1 */
70          else send U2 = (4, ⌊(n0 + n1)/10k⌋, d = 0, V = {i}, i);
71          repeat
72                  if receiving R = (1, r) then buffer R fi;
73                  if receiving A = (2, r, s, n, op(s))
74                    then if r = i or r is between i and s then kill A /* old message or r is off-
                      line */
75                                                  else buffer A
76                          fi
77                  fi;
78                  if receiving an U1 message then kill U1;
79                  /* it cannot receive U2 and U3 for r ≠ i */
80                  fi
81          until receiving U2 = (4, ⌊(n0 + n1)/10k⌋, d, V, i);
82          send U3 = (5, n = n0 + n1, V, i);
83          repeat
84                  if receiving R = (1, r) then buffer R fi;
85                  if receiving A = (2, r, s, n, op(s))
86                    then if r = i or r is between i and s then kill A /* old message or r is off-
                      line */
87                                                  else buffer A
88                          fi
89                  fi;
90                  if receiving an U1 message then kill U1
91                  /* it cannot receive U2 and U3 for r ≠ i */
92                  fi
93          until receiving U3 = (5, n, V, i);
94          get n;
95          get op(i) from V;
96    fi;
97    2: reply to all buffered A and R messages with A = (2, r, i, n, op(i));
98    /* i.e., eventually substitute the old A received */
99    become active;
100 else /* i is active */
101 repeat
102         if receiving R = (1, r)
103         then if a = 0 and r ≠ i then send A(2, r, i, n, op(i)) back
104                                 else kill R /* r will get its values anyway */
105               fi
```

```
106        fi;
107        if receiving A = (2, r, s, n, op(s))
108          then if r is between i and s or r = i
109                  then kill A /* r went off-line or A is an old message for i*/
110                  else forward A = (2, r, i, n, op(i))
111               fi
112        fi;
113        if receiving a message U1 = (3, n0, n1, r)
114          then a := 1;
115               forward U1 = (3, n0 + 1, n1, r)
116               /* r cannot go off-line */
117        fi;
118        if receiving a message U2 = (4, ⌊(n0 + n1)/10k⌋, d, V, r)
119          then if ⌊(n0 + n1)/10k⌋ = d − 1
120                  then forward U2 = (4, ⌊(n0 + n1)/10k⌋, 0, V ∪ {i}, r)
121                  else forward U2 = (4, ⌊(n0 + n1)/10k⌋, d + 1, V, r)
122               fi
123        fi;
124        if receiving a message U3 = (5, n, V, r)
125          then get n;
126               get op(i) from V;
127               a := 0;
128               forward U3
129        fi;
130   until off-line
131   fi.
```

References

[1] Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science* (*FOCS*), pages 370–375, 30 October to 1 November 1989.

[2] B. Awerbuch. Shortest paths and loop-free routing in dynamic networks. *SIGCOMM Computer Communication Review*, 20(4):177–187, September 1990.

[3] B. Awerbuch, A. Bar-Noy, D. Peleg, and N. Linal. Improved routing strategies with succinct tables. *Journal of Algorithms*, 11(3):307–341, September 1990.

[4] B. Awerbuch and Y. Mansour. An efficient topology update protocol for dynamic networks. In *Proceedings of the 6th International Workshop on Distributed Algorithms* (*WDAG*), pages 185–202, November 1992.

[5] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks. In *Proceedings of the 24th ACM Symposium on Theory of Computing* (*STOC*), pages 557–570, May 1992.

[6] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilizing end-to-end communication. *Journal of High Speed Networks*, 5(4):365–381, 1996.

[7] E.M. Bakker, J. van Leeuwen, and R.B. Tan. Linear interval routing. *Algorithms Review*, 2(2):45–61, 1991.

[8] E.M. Bakker, J. van Leeuwen, and R.B. Tan. Prefix routing schemes in dynamic networks. *Computer Networks and ISDN Systems*, 26(4):403–421, December 1993.

[9] S. Dolev, E. Kranakis, D. Krizanc, and D. Peleg. Bubbles: adapting routing scheme for high-speed dynamic networks. *SIAM Journal on Computing*, 29(3):804–833, December 1999.

[10] M. Flammini, G. Gambosi, and S. Salomone. Interval routing schemes. *Algorithmica*, 16(6):549–568, December 1996.

[11] P. Fraigniaud and C. Gavoille. A characterization of networks supporting linear interval routing. In *Proceedings of the* 13*th ACM Conference on Principles of Distributed Computing* (*PODC*), pages 216–224, August 1994.
[12] P. Fraigniaud and C. Gavoille. Optimal interval routing. In *Proceedings of Parallel Processing*: *CONPAR* '94, pages 785–796. LNCS 854. Springer-Verlag, Berlin, September 1994.
[13] P. Fraigniaud and C. Gavoille. Interval routing schemes. *Algorithmica*, 21(2):155–182, June 1998.
[14] C. Gavoille. Lower bounds for interval routing on bounded degree networks. Technical Report RR-1144-96, LaBRI University of Bordeaux, France, October 1996.
[15] C. Gavoille. A survey on interval routing scheme. *Theoretical Computer Science*, 245(2):217–253, August 2000.
[16] E. Kranakis, D. Krizanc, and S.S. Ravi. On multi-label linear interval routing schemes. *The Computer Journal*, 39(2):133–139, 1996.
[17] D. Krizanc, F.L. Luccio, and R. Raman. Dynamic interval routing on asynchronous rings. In *Proceedings of the IEEE* 13*th International Parallel Processing Symposium and* 10*th Symposium on Parallel and Distributed Processing* (*IPPS/SPDP*), pages 225–232, April 1999.
[18] J. van Leeuwen and R.B. Tan. Interval routing. *The Computer Journal*, 30(4):298–307, August 1987.
[19] L. Narayanan and N. Nishimura. Interval routing on k-trees. *Journal of Algorithms*, 26(2):325–369, February 1998.
[20] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, July 1989.
[21] N. Santoro and R. Khatib. Labeling and implicit routing in networks. *The Computer Journal*, 28(1):5–8, February 1985.
[22] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall, Engelwood Cliffs, NJ, 1996.

Received April 5, 2002, *and in revised form January* 7, 2003, *and April* 4, 2003, *and in final form April* 14, 2003. *Online publication May* 5, 2004.