Compact Routing Schemes for Dynamic Ring Networks *

Danny Krizanc
Mathematics Department,
Wesleyan University, Middletown, CT, U.S.A.
dkrizanc@wesleyan.edu

Flaminia L. Luccio†
Dipartimento di Scienze Matematiche,
Università degli Studi di Trieste, Italy
luccio@mathsun1.univ.trieste.it

Rajeev Raman
Department of Mathematics and Computer Science,
University of Leicester, Leicester, 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, 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 stretch $\frac{N}{2}$, the other with adaptation cost $O(N)$ messages of $O(\log N)$ bits and stretch 1. The third algorithm is randomized, uses update messages of size $O(k \log N)$, has adaptation cost $O(k)$ and expected stretch $1 + \frac{1}{k}$, for any $k \geq 1$. All schemes require $O(\log N)$ bits per node for the routing information and all messages headers are of $O(\log N)$ bits.

*Research supported by NSERC grant and by MIUR. Rajeev Raman's work was supported in part by EPSRC grant GR/L92150. A preliminary version of this paper has been presented at the *IEEE 13-th International Parallel Processing Symposium and 10-th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99)* [17].

†Correspondence may be sent to Flaminia L. Luccio, c/o Dipartimento di Scienze Matematiche, Università degli Studi di Trieste, via Valerio 12/1, 34127 Trieste, phone: 0039/040-5582610, fax: 0039/040-5582623.

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, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21]. Most of the research done in this area has concerned static networks and has focussed 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, 5, 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 amortized number of messages exchanged per topology change, for dynamic routing schemes.

An important example of static routing schemes are k-Interval Routing Schemes ($k - IRS$s) [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 labelled 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 proportional to $kd \log N$ 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 ($DIRS$s). A $DIRS$ for a network with maximum size N is based on the IRSs: nodes are labelled by the set $\{0, \ldots, N-1\}$ and arcs are labelled by disjoint intervals from 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. Moreover, an update procedure is defined in order to dynamically change the routing tables, i.e., the range of the intervals assigned to the arcs, whenever a change occurs in the network.

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, *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]).

The *correctness* of the message delivery of a routing scheme is based upon two factors:

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 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 dynamically changing ring of processors with FIFO queues and with global orientation. We assume the maximum size of the ring is N, and every possible position in the ring has a unique label from the set $\{0, \dots, N-1\}$. Messages move between switches each of which is connected to a processor. 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. In order to ensure correctness we assume that there is a unique processor, the one labeled with value 0, that is always active. We will 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. (See figure 1.) This model captures the common star-shaped ring LAN topology in which all connections pass through a wire center as well as rings consisting of virtual links in optical networks.

Below we describe three different DIRSs for the ring which show a tradeoff between the stretch, 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 $N/2$, the other with worst-case adaptation cost $O(N)$ messages of $O(\log N)$ bits but with stretch 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 $1+1/k$, for any integer $k \geq 1$. 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.

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 polylogarithmic stretch.

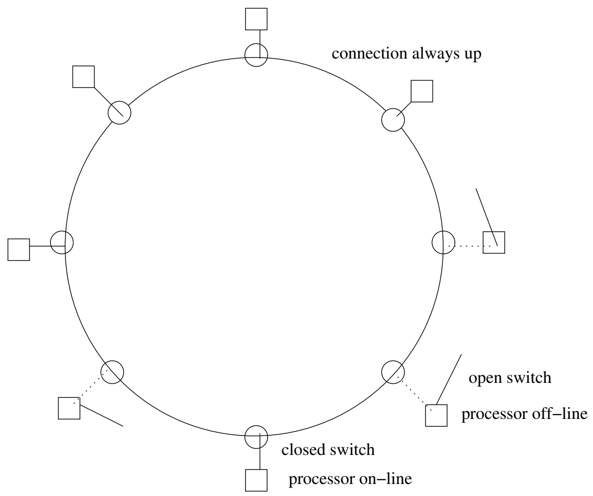

Figure 1: Switches on and off.

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 performing updates) but has worst-case stretch $\lfloor \frac{N}{2} \rfloor$.

We assume the classical node and arc labeling of the IRS for a ring of size N [18, 21]. A message $M = (D, r, s)$ contains the data D, the name of the receiver r and of the sender s.

The $DIRS$ consists of the following two algorithms: an Update procedure executed by a processor i that goes on-line that using the classical optimal stretch IRS associates a fixed interval both to the left and to the right arc and becomes active; a Routing procedure for active and pending processors. Active processors may send messages using the classical optimal stretch IRS, i.e., by checking if the label of the destination node is in the interval of the right or left arc; both active and pending processors may receive a message but they kill it if the destination went off-line, they forward it if it is not destined for them or they process it if they are the destination (pending processors kill it in this case). Appendix A contains the The Update and Routing procedure for the $DIRS$ (algorithm 1 and 2 respectively). Note that no action is required of a processor going off-line.

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 is at most $\lfloor \frac{N}{2} \rfloor$.*

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 assumed to be always active.)

The space complexity is $O(\log N)$ bits since every processor stores 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 what is the sequence of changes 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 $j = \lfloor i + \frac{N}{2} \rfloor \bmod N$. This trivially implies that at any time the stretch can be at most $\frac{\lfloor i + \frac{N}{2} - i \rfloor \bmod N}{1} = \lfloor \frac{N}{2} \rfloor$ since a message travels always in the same direction and since at most $\lfloor \frac{N}{2} \rfloor$ processors can be active along that path. ■

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 sequence of m changes to the network (processors going on-line or off-line), the amortized adaptation cost is $O(n)$ update messages of $O(\log N)$ bits where n is the maximum number of active processors during the sequence. Note that in the worst case $n = N$ and the adaptation cost is $O(N)$.

3.1 The Algorithm

The routing scheme we use is the classical optimal stretch IRS where each processor maintains the value of the label of the processor that is precisely opposite to it, i.e., there are the same (to within 1) number of processors on a left path or right path to that processor. This value is dynamically updated by sending a constant number of messages of size $O(\log N)$ all the way around the ring, incurring a cost of at most $O(n)$. 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.

The algorithm is divided into a Routing procedure and an Update procedure. The Routing procedure (algorithm 3 in the appendix) is the classical optimal stretch IRS: an active processor s that wants to send a data message $M = (0, D, r, s)$ containing the data D to a processor r, checks if $s + 1 \leq r \leq op(s)$ (where $op(s)$ is the name of the opposite processor, with equal, to within 1, number of active processors in both directions) and if so sends the message to the left, else to the right. (The computations are done $\bmod N$.) Every active processor i that receives the data message $M = (0, D, r, s)$ either processes it (if $i = r$), or forwards it unless it finds out that the destination processor went off-line (i.e., r is between i and the sender s in the direction the message came from) and in this case it kills the message. Every pending processor j either forwards the data message or kills it if the destination went off-line or if $r = j$ (i.e., j is the receiver).

The update strategy (algorithm 4 in the appendix) is more complicated. Every processor i that goes on-line becomes pending and gets its fixed label i. It then starts an update by sending a Phase 1 message in the direction of the orientation (e.g., to the left). This message collects the value of its left and right neighbours (l^i and r^i respectively) and their opposite values ($op(l^i)$ and $op(r^i)$ respectively). If i gets the message back 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 another Phase 1 message it lets it through only if the sender's label is bigger. Every processor in higher phases stops and buffers in a FIFO queue Phase

1 messages. It is easy to see that a unique processor (the maximum among pending processors) moves to Phase 2. Let us assume i is the unique processor that moves to Phase 2 and 3. In the next two phases i sends two update messages around in both directions containing the name of the previous active processor and its opposite value (i.e., first r^i and $op(r^i)$, and then l^i and $op(l^i)$ or vice versa, depending on the assumed oddness or evenness of the ring), so that the other processors in the ring can update their opposite values if necessary. This is possible since while a single insertion occurs every processor will have as an opposite value the opposite value of its right or left neighbour depending on the evenness or oddness of the ring. During this execution i buffers messages from processors in Phase 1. When i completes the updates it becomes active and it lets all these messages through so that a the rest of the updates can eventually start. This mechanism essentially sequentializes all insertions.

The complete code for the Routing and Update procedure for the DIRS are given in the appendix.

3.2 Algorithm Analysis

Theorem 2. *The $DIRS$ defined by algorithm 3 and 4 modified also for the off-line case is correct and has the following properties:*

1. *the stretch is 1;*
2. *the space required for the routing tables is at most $O(\log N)$ bits per node;*
3. *the amortized adaption cost for a sequence of m changes to the network is $O(n)$ messages of $O(\log N)$ bits each, where n is the maximum number of active or pending processors during the sequence.*

PROOF The general correctness of the Routing procedure derives from the fact that the $DIRS$s 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 right 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. Note that processor 0 is always active. On the other hand if r is on-line it will eventually receive the message and in the case it is pending it will kill it. Therefore all data messages are correctly delivered. 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). Let us assume by contradiction that at least 2 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. W.l.o.g., let us 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 into 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 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 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 neighbour since no other processor is in Phase 2 or 3. On the other hand let us assume that as M gets to another processor j with a bigger value, it finds that j is doing an update. M will stop at j and i will eventually get it after, but the correct values of its neighbours might have changed after j's update. 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 neighbours and $op(r^x)$, $op(l^x)$ their opposite values respectively, a boolean variable $even(x)$ set to 1 if the number of processors known by x in the ring is even, to 0 otherwise (up to the last update it has seen) as well as the values of $even(r^x)$ and $even(l^x)$. Therefore i will have its values updated. 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 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 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 at every step the size of the ring is at most n and every update requires at most three messages to travel around the ring. Therefore the amortized cost for a sequence of m insertions and deletions is $O(mn/m) = O(n)$. Messages are $O(\log N)$ bits each. ■

4 Scheme with constant expected adaptation cost

In this section we describe a randomized $DIRS$ that at quiescence routes with expected stretch $1 + 1/k$ and requires an expected amortized $O(k)$ messages containing $O(k \log N)$ bits each for each change in the ring, for any $k \geq 1$. 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 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 $[op(i), (i-1) \bmod N]$, 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 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 (algorithm 3 in the appendix) with the only difference that in this case $op(i)$ is an estimate of i's true opposite value. We use the classical IRS schemes with the added feature that messages destined for off-line processors are removed from the system: Every active processor either processes or forwards the data message in the direction defined by the related interval unless it concludes that the destination processor is off-line (i.e., the message has passed the position of the destination and has not been stopped) in which case it kills the message. Every pending processor forwards the data message in the direction it has been moving or kills it if it concludes the destination is off-line or if the destination is itself (since it is an old message).

The Update procedure (algorithm 5 in the appendix) is more complicated. Every pending processor has to do the following things: it sends a request message R asking the next active processor j its estimated n value and $op(j)$. It waits for an answer message A and it flips a coin with probability of heads equal to $\min\{1, \frac{10k}{n}\}$. If it gets a tail it becomes active and uses $op(j)$ for its own opposite value and n as its estimate of the size of the ring. Otherwise, it starts an update phase similar to that of algorithm 4 by sending a Phase 1 message that counts the number of active and pending processors. 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 with a count of the number of active processors, n_0, and pending processors, n_1. It then sends a Phase 2 message that collects every $\lfloor \frac{n_0+n_1}{10k} \rfloor$th processor label of nodes that are active or pending during Phase 1. Finally during Phase 3 it sends an update message containing the labels it gathered in Phase 2 and every active and pending processor updates its opposite value, as well as their estimate of n. A pending processor awaiting an answer A message that receives a Phase 1 message is counted in n_1 and gets activated at the end of Phase 3 without flipping a coin.

A pending processor awaiting an answer A message that receives a Phase 2 or Phase 3 message waits until the completion of the update at which point it will have received the new n estimate and opposite value and it will flip a coin and eventually start a new update if the result is heads. Whenever it receives its A message it kills it.

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.

The procedure for a processor going off-line is analogous to the above. It flips a coin with probability $\min\{1, \frac{10k}{n}\}$ 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 it moves to Phase 2, processor labels are collected and in Phase 3, the opposite values of active processors are updated. At the completion of Phase 3, it 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.

The $DIRS$ consists of the Routing and Update procedures whose code is given in the appendix.

4.2 Algorithm Analysis

Theorem 3. *For any $k \geq 1$, the randomized $DIRS$ defined by algorithm 3 and 5 modified also for the off-line case is correct and has the following properties:*

1. *the expected stretch is at most $1 + \frac{1}{k}$;*
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 become active or go 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 of $1 + \frac{1}{k}$. 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 Phase 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 $\min\{1, \frac{10k}{n_0+n_1}\}$. 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 $\frac{n_0+n_1}{10k}$.

To compute the stretch there are two cases depending on whether n_2 is positive or negative. The idea is the following: every processor maintains an opposite value that is correct but for an interval of $\frac{n_0+n_1}{10k}$ processors. Moreover the actual number of on-line processors is $n_0 + n_1 + n_2$ (at quiescence) and in the worst case n_2 are either added to the side of the path taken between two points or subtracted from the side of the path not taken. In the worst case a processor wants to send a message to a processor in the exact opposite block of $\frac{n_0+n_1}{10k}$ processors. If n_2 is positive then for any $k \geq 1$ the stretch is bounded by:

$$\frac{\frac{(5k+1)(n_0+n_1)}{10k} + \frac{n_0+n_1}{10k}}{\frac{(5k-1)\,(n_0+n_1)}{10k}} \leq 1 + \frac{1}{k}.$$

If n_2 is negative then for any $k \geq 1$ the stretch is bounded by:

$$\frac{\frac{(5k+1)(n_0+n_1)}{10k}}{\frac{(5k-1)\,(n_0+n_1)}{10k} - \frac{n_0+n_1}{10k}} \leq 1 + \frac{1}{k}.$$

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, and Phase 2 and 3 messages (U_2 and U_3) have at most $3 + (10k + 3)\log N$ bits.

The expected number of messages sent per update can be bounded as follows. A pending processor is responsible for the sending of 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 doesn't, plus the Phase 1 message of the eventual "winner" of the Phase 1 "race", and at most one Phase 2 and Phase 3 messages. After flipping its coin, with probability of heads equal to $\min\{1, \frac{10k}{n}\}$, 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_0 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 $3(n+n_0)$ messages for each of the three phases. During this period $1+n_0$ 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

$$6 + \min\{1, \frac{10k}{n}\}\frac{3(n+n_0)}{1+n_0} = O(k).$$

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 stretch $\frac{N}{2}$, the other with adaptation cost $O(N)$ of size $O(\log N)$ bits and stretch 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 $1 + \frac{1}{k}$. 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.

References

[1] Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *Proceedings of the 30-th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 370–375, 30 October-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, and D. Peleg N. Linal. Improved routing strategies with succint 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 6-th 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 24-th 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, 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, 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*, Springer Verlag LNCS 854, pages 785–796, September 1994.

[13] P. Fraigniaud and C. Gavoille. Interval routing schemes. *Algorithmica*, 21(2):155–182, 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, 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.

A Appendix

Algorithm 1 /* Update procedure for processor i */

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

Algorithm 2 /* Routing procedure for processor i*/

repeat
    if active **and** willing to send a message M to a node r **then**
      if $(i+1) \bmod N \leq r \leq \lfloor i + \frac{N}{2} \rfloor \bmod N$
        then send $M = (D, r, i)$ to the left
        else to the right
      fi
    fi;
    if receiving $M = (D, r, s)$ **then**
      if $r \neq i$ **then**
            if r is not between s and i
              then forward M
              else kill M /* r went off-line */
            fi
          else /* $r = i$ */
            if pending **then** kill M
                else process M
            fi
      fi
    fi
until off-line;

Algorithm 3 /* Routing procedure for processor i */

if pending
  then if receiving a data message $M = (0, D, r, s)$
      then if r is not between s and i **and** $r \neq i$
          then forward $M = (0, D, r, s)$
          else kill M /* r went off-line or M is an old message */
        fi

```
        fi
  else  /* it is active */
        if sending a data message M = (0, D, r, i) to a processor r
          then if (i + 1) mod N ≤ r ≤ op(i)
                   then send M = (0, D, r, i) to the left
                   else to the right
                fi
          else if receiving M = (0, D, r, s)
                  then if r ≠ i
                          then if r is not between s and i
                                 then forward M = (0, D, r, s)
                                 else kill M /* r went off-line */
                               fi
                          else /* r = i */
                               process M
                       fi
                fi
        fi
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 a and b will be at the end 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 neighbour 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 $M = (3, r, k, op(k), even(k))$. */

```
if pending then
  /* the new pending processor has to get new values */
  get value i;
  r^i, l^i, op(i), op(l^i), op(r^i), even(i), even(r^i), even(l^i), oldop(i) := −1;
  send M = (1, i, −1, −1, −1, −1, −1, −1) to the left;
  /* sending a new update Phase 1 message */
  repeat
        if receiving M = (1, r, a, op(a), even(a), b, op(b), even(b))
          /* another processor r ≠ i is in Phase 1 */
          then if i > r
                  then buffer M
                  else forward M
               fi
          else if receiving M = (2, r, op(r), prev(r), k, op(k), even(k), oldop(k)) or
                  M = (3, r, k, op(k), even(k)) /* i receives a Phase 2 or 3 message */
```

```
            then /* $i$ updates left or right values since some other
                  processor has gone on-line before it */
                  if receiving $M$ from the left
                     then $l^i := k$;
                           $op(l^i) := op(k)$;
                           $even(l^i) := even(k)$
                     else $r^i := k$;
                           $op(r^i) := op(k)$;
                           $even(r^i) := even(k)$
                  fi;
                  forward $M$
         fi
      fi
until receiving $M = (1, i, a, op(a), even(a), b, op(b), even(b))$;
/* $i$ got back its Phase 1 message */
if $M = (1, i, -1, -1, -1, -1, -1, -1)$
/* $i$ is the only processor on-line */
   then $even(i) := 0$;
         goto 1
   else if $l^i = -1$ then /* $i$'s left values have not been updated yet */
                        $l^i := a$;
                        $op(l^i) := op(a)$;
                        $even(l^i) := even(a)$
      fi;
      if $r^i = -1$ then /* $i$'s right values have not been updated yet */
                        $r^i := b$;
                        $op(r^i) := op(b)$;
                        $even(r^i) := even(b)$
      fi;
      /* asking the right neighbour for the oddness or evenness */
      if $even(r^i) = 1$ /* $i$ gets values of right neighbour */
         then $op(i) := op(r^i)$;
               $even(i) := \neg even(r^i)$
         else $op(i) := op(l^i)$;
               $even(i) := \neg even(l^i)$
      fi;
      /* a Phase 2 message has to be sent */
      if $even(i) = 1$
         then $prev(i) := l^i$;
               send $M = (2, i, op(i), prev(i), i, op(i), even(i), oldop(i))$ to the right
         else $prev(i) := r^i$;
              send $M = (2, i, op(i), prev(i), i, op(i), even(i), oldop(i))$ to the left
      fi;
      repeat
```

```
            if receiving M with r ≠ i then buffer M fi
        until receiving M = (2, i, op(i), prev(i), k, op(k), even(k), oldop(k));
        /* k is the previous active processor */
        if receiving M from the right
          then /* update right values */
                op(r^i) := op(k);
                even(r^i) := even(k)
          else /* update left values */
                op(l^i) := op(k);
                even(l^i) := even(k)
        fi;
        M := (3, i, i, op(i), even(i));
        if even(i) = 1
          then send M to the left
          else to the right
        fi;
        repeat
            if receiving M with r ≠ i then buffer M fi
        until receiving M = (3, i, k, op(k), even(k));
        if receiving M from the right
          then /* update right values */
                op(r^i) := op(k);
                even(r^i) := even(k)
          else /* update left values */
                op(l^i) := op(k);
                even(l^i) := even(k)
        fi;
        if l^i = r^i then /* there is only another active processor */
                        oldop(i) := op(i);
                        op(i) := l^i;
                        op(l^i) := op(r^i) := i
        fi
    fi;
1:  process buffered messages in FIFO order and eventually update their old values;
    let messages eventually go through;
    become active
  else /* i is active */
  repeat
        if receiving M = (1, r, a, op(a), even(a), b, op(b), even(b)) /* Phase 1 message */
          then if r = i then kill M /* r cannot go off-line */
                         else /* M has to be updated */
                              if a = op(a) = even(a) = −1
                                then M := (1, r, i, op(i), even(i), i, op(i), even(i))
                                      /* i is r's left processor */
```

else $M := (1, r, a, op(a), even(a), i, op(i), even(i))$
/*i is r's possible right processor */
fi;
forward M
fi
else if receiving $M = (2, r, op(r), prev(r), k, op(k), even(k), oldop(k))$
/* Phase 2 message */
then if coming from the right
then/* update right and local values */
$r^i := k$;
$op(r^i) := op(k)$;
$even(i), even(r^i) := even(k)$;
if $k = op(r)$ /* the previous active processor is $op(r)$ */
then $oldop(i) := op(i)$;
$op(i) := r$
else if $i \in (op(r), prev(r)]$
/* i is between $op(r)$ and r */
then $oldop(i) := op(i)$;
$op(i) := oldop(k)$
fi
fi
else /* update left and opposite values */
$l^i := k$;
$op(l^i) := op(k)$;
$even(i), even(l^i) := even(k)$;
if $i = op(r)$ /* i is opposite to r */
then $oldop(i) := op(i)$;
$op(i) := r$
else if $i \in [prev(r), op(r))$
/* i is between r and $op(r)$ */
then $oldop(i) := op(i)$;
$op(i) := oldop(k)$
fi
fi
fi;
forward $M = (2, r, op(r), prev(r), i, op(i), even(i), oldop(i))$
else if receiving $M = (3, r, k, op(k), even(k))$
/* Phase 3 message */
then if coming from the right
then/* update right values */
$r^i := k$;
$op(r^i) := op(k)$;
$even(r^i) := even(k)$
else/* update left values */

$l^i := k$;
$op(l^i) := op(k)$;
$even(l^i) := even(k)$
fi;
if $r^i = l^i$ **then** $oldop(i) := op(i)$;
$op(i) := r^i$;
$op(r^i) := op(l^i) := i$
fi;
forward $M = (3, r, i, op(i), even(i))$
fi
fi
fi
until off-line
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 $U_2 = (4, \frac{n_0+n_1}{10k}, d, V, r)$, where d is a counter and V a vector of processors labels; finally Phase 3 messages are in the form $U_3 = (5, \frac{n_0+n_1}{10k}, d, V, r)$. */
if pending **then**
get value i;
$a, exit := 0$;
send $R = (1, i)$ to the left;
/* this is a request of the n and $op(next(i))$ values */
repeat
if receiving a message $U_1 = (3, n_0, n_1, r)$
then $a := 1$ /* message A for i will have to be ignored and the update will be global */ ;
forward $U_1 = (3, n_0, n_1 + 1, r)$
fi;
if receiving a message $U_2 = (4, \frac{n_0+n_1}{10k}, d, V, r)$
then if $a = 1$ /* i may have to be added to V */
then if $\lfloor \frac{n_0+n_1}{10k} \rfloor = d$
then forward $U_2 = (4, \frac{n_0+n_1}{10k}, 0, V \bigcup \{i\}, r)$
else forward $U_2 = (4, \frac{n_0+n_1}{10k}, d + 1, V, r)$
fi
else /* $a = 0$ */
forward $U_2 = (4, \frac{n_0+n_1}{10k}, d, V, r)$
fi

fi;
if receiving a message $U_3 = (5, \frac{n_0+n_1}{10k}, d, V, r)$
then get n;
get $op(i)$ from V;
$exit := 1$;
forward U_3
fi;
if receiving $R = (1, r)$ **then** buffer R **fi**;
if receiving $A = (2, r, s, n, op(s))$ **and** $r \neq i$
then if r is between i and s **then** kill A
else buffer A
fi;
if $(a = 1)$ **and** receiving $A = (2, i, s, n, op(s))$ **then** kill A **fi**
until $exit$ **or** $(a = 0$ **and** receiving $A = (2, i, s, n, op(s))$);
if not $exit$ **then** get new n and $op(i)$ value from A;
if $((a = 1)$ **and** $exit)$ **then goto 2** /* i was pending during Phase 1 */
else /* i was pending during Phase 2 or 3 or received A */
1: flip the coin;
$a, exit := 0$;
if head /* obtained with probability $\min\{1, \frac{10k}{n}\}$ */
then send an update message $U_1 = (3, 0, 0, i)$
repeat
if receiving $A = (2, r, s, n, op(s))$
then if $r = i$ **or** r is between i and s **then** kill A /* old message or r is off-line */
else buffer A
fi
fi;
if receiving $R = (1, r)$ **then** buffer R **fi**;
if receiving $U_1 = (3, n_0, n_1, r)$ **and** $r \neq i$
then if $r < i$ **then** kill U_1
else forward $U_1 = (3, n_0, n_1 + 1, r)$;
$a := 1$
fi
fi;
if receiving $U_2 = (4, \frac{n_0+n_1}{10k}, d, V, r)$
then if $a = 1$ **then if** $\lfloor \frac{n_0+n_1}{k} \rfloor = d$
then forward $U_2 = (4, \frac{n_0+n_1}{10k}, 0, V \bigcup \{i\}, r)$
else forward $U_2 = (4, \frac{n_0+n_1}{10k}, d + 1, V, r)$
fi
else forward $U_2 = (4, \frac{n_0+n_1}{10k}, d, V, r)$ /* $a = 0$ */
fi
fi;
if receiving $U_3 = (5, \frac{n_0+n_1}{10k}, d, V, r)$
then get n;

```
                get op(i) from V;
                exit := 1;
                forward U_3
            fi
    until receiving U_1 = (3, n_0, n_1, i) or exit;
    if exit = 1 and a = 1 then goto 2 /* i was pending during Phase 1 */
    else send U_2 = (4, (n_0+n_1)/10k, d = 0, V, i);
    repeat
        if receiving R = (1, r) then buffer R fi;
        if receiving A = (2, r, s, n, op(s))
          then if r = i or r is between i and s then kill A /* old message or r is off-line */
                                                   else buffer A
             fi
        fi;
        if receiving an U_1 message then kill U_1;
        /* it cannot receive U_2 and U_3 for r ≠ i */
        fi
    until receiving U_2 = (4, (n_0+n_1)/10k, d, V, i);
    send U_3 = (5, (n_0+n_1)/10k, d = 0, V, i);
    repeat
        if receiving R = (1, r) then buffer R fi;
        if receiving A = (2, r, s, n, op(s))
          then if r = i or r is between i and s then kill A /* old message or r is off-line */
                                                   else buffer A
             fi
        fi;
        if receiving an U_1 message then kill U_1
        /* it cannot receive U_2 and U_3 for r ≠ i */
        fi
    until receiving U_3 = (5, (n_0+n_1)/10k, d, V, i);
    get n;
    get op(i) from V;
  fi;
  2: reply to all buffered A and R messages with A = (2, i, s, n, op(i));
  /* i.e., eventually substitute the old A received */
  become active;
else /* i is active */
repeat
    if receiving R = (1, r)
    then if a = 0 and r ≠ i then send A(2, r, i, n, op(i)) back
                             else kill R /* r will get its values anyway */
       fi
    fi;
    if receiving A = (2, r, s, n, op(s))
```

```
            then if r is between i and s and r = i
                    then kill A /* r went off-line or A is an old message for i*/
                    else forward A = (2, r, i, n, op(i))
                  fi
          fi;
          if receiving a message U_1 = (3, n_0, n_1, r)
            then a := 1;
                  forward U_1 = (3, n_0 + 1, n_1, r)
                  /* r cannot go off-line */
          fi;
          if receiving a message U_2 = (4, (n_0+n_1)/10k, d, V, r)
            then if ⌊(n_0+n_1)/10k⌋ = d
                    then forward U_2 = (4, (n_0+n_1)/10k, 0, V ⋃{i}, r)
                    else  forward U_2 = (4, (n_0+n_1)/10k, d + 1, r)
                  fi
          fi;
          if receiving a message U_3 = (5, (n_0+n_1)/10k, d, V, r)
            then get n;
                  get op(i) from V;
                  a := 0;
                  forward U_3
          fi;
    until off-line
fi.
```