On a Space-optimal Distributed Traversal Algorithm[*]

Sébastien Tixeuil

Laboratoire de Recherche en Informatique, UMR CNRS 8623,
Université de Paris Sud, 91405 Orsay cedex, France.
email: `tixeuil@lri.fr`

Abstract

A traversal algorithm is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is Eulerian if every edge is examined exactly once. We present a simple deterministic distributed algorithm for the Eulerian traversal problem that is space-optimal: each node has exactly d states, where d is the outgoing degree of the node, yet may require $O(m^2)$ message exchanges before it performs an Eulerian traversal, where m is the total number of edges in the network. In addition, our solution has failure tolerance properties: *(i)* messages that are exchanged may have their contents corrupted during the execution of the algorithm, and *(ii)* the initial state of the nodes may be arbitrary.

Then we discuss applications of this algorithm in the context of self-stabilizing virtual circuit construction and cut-through routing. Self-stabilization ([6, 7]) guarantees that a system eventually satisfies its specification, regardless of the initial configuration of the system. In the cut-through routing scheme, a message must be forwarded by intermediate nodes before it has been received in its entirety. We propose a transformation of our algorithm by means of randomization so that the resulting protocol is self-stabilizing for the virtual circuit construction specification. Unlike several previous self-stabilizing virtual circuit construction algorithms, our approach has a small memory footprint, does not require central preprocessing or identifiers, and is compatible with cut-through routing.

1 Introduction

Traversal A *traversal* algorithm is a systematic procedure for exploring a graph by examining all of its vertices and edges. Traversal algorithms are typically used to explore unknown graphs (see [5]) and build a map as the graph is visited. The case of Eulerian directed graphs (where every node has as many incoming edges as outgoing edges) offers best performance since a traversal may be performed by visiting each link exactly once (it is then an *Eulerian traversal*). In [5], a centralized algorithm is proposed that traverses an Eulerian graph by visiting at most $2m$ edges, where m is the overall number of directed edges; once the graph map has been built, one can easily use well-known centralized algorithms to compute an *Eulerian cycle* (a cycle that includes all edges

[*]This a joint submission to DISC'01 and WSS'01

exactly once), which in turn can be used to perform an Eulerian traversal. In [10], a distributed solution to the Eulerian cycle construction is given, that requires $2m$ message exchanges and $\Omega(d^2)$ memory states at each node, where d is the outgoing degree of the node.

Self-stabilization Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient faults because they are exposed to constant change of their environment. One of the most inclusive approaches to fault tolerance in distributed systems is *self-stabilization* [6, 7]. Introduced by Dijkstra in [6], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior or the set of *legitimate* states. Since most self-stabilizing fault-tolerant protocols are non-terminating, if the distributed system is subject to transient faults corrupting the internal node state but not its behavior, once faults cease, the protocols themselves guarantee to recover in a finite time to a safe state without the need of human intervention. This also means that the complicated task of initializing distributed systems is no longer needed, since self-stabilizing protocols regain correct behavior regardless of the initial state. Furthermore, note that in practice, the context in which we may apply self-stabilizing algorithms is fairly broad since the program code can be stored in a stable storage at each node so that it is always possible to reload the program after faults cease or after every fault detection.

Cut-through routing The *cut-through* routing is used in many ring networks (including IBM *Token Ring* and *FDDI*). In this routing scheme, a node can start forwarding any portion of a message to the next node on the message's path before receiving the message in its entirety. If this message is the only traffic on the path, the total delay incurred by the message is bounded by the transmission time (calculated on the slowest link on the path) plus the propagation delay. So, the total message delay is proportional to the length of the message and to the number of links on the path. Some pieces of the same message may simultaneously be traveling on different links and some other pieces are stored at different nodes. As the first bit of the message is transmitted on the links on the message's routing path, the corresponding links are reserved, and the reservation of a link is released when the last bit of the message is transmitted on the link.

This approach removes the need of having a local memory of any node greater than the one required to store a bounded number of bits, and also reduces the message delay to a small (bounded by the buffer size of the node) value. As with the current processors, the time needed for sending/receiving bits to/from a communication medium is far greater than the time needed to perform the basic computational steps (such as integer calculations, tests, read/write from/to registers, etc.), we can assume that a given process can perform a limited number of steps between the receipt of two pieces of a message.

Our contribution First, we present a distributed algorithm that is state optimal relatively to the Eulerian traversal problem. At every node, exactly d memory states are needed, where d is the actual outgoing degree of the node. In addition, our algorithm makes very little hypothesis about the system on which it is run. For example, nodes need not have unique identifiers or a special distinguished leader. Node variables need not be properly initialized when the protocol is started.

Moreover, our protocol remains behaving accordingly to the Eulerian traversal specification even when messages that are exchanged between nodes have their content arbitrarily corrupted from time to time, even during the execution of the algorithm. Still, when it is first started, our algorithm may exhibit a transient $O(m^2)$ time period (where m is the overall number of edges) during which it performs the first traversal of the network. That first traversal may not be Eulerian, but every subsequent traversal is and remains Eulerian.

Second, and hinted by the failure tolerance properties of our algorithm (message contents corruption, node memory initial corruption), we transform it into a self-stabilizing virtual circuit construction algorithm by means of randomization. Informally, there exist two kinds of self-stabilizing virtual circuit constructions in the literature. Some (as reported in [7]) assume bidirectional networks (which are a proper subset of Eulerian networks), and construct in a self-stabilizing way a spanning tree, then performs an Eulerian tour of this tree (which is trivially done). Others (*e.g.* [1, 12]) assume only strongly connected networks (which are a proper superset of Eulerian networks), but either require some central preprocessing or unique node identifiers, have high memory footprint, and are not compatible with cut-through routing. In comparison, our virtual circuit construction only performs in directed Eulerian networks, yet does not require central preprocessing or identifiers, has small memory footprint, and is compatible with cut-through routing. Moreover, when used as a lower layer by some other algorithm using the composition scheme of *e.g.* [8], the so-constructed virtual ring permits a bijective mapping between nodes in the original Eulerian network and nodes in the virtual ring network. This bijection permits not to change the upper layer algorithm. Then, previously known self-stabilizing cut-through algorithms that perform on unidirectional rings (*e.g.* [2, 4, 14]) can now be run on Eulerian networks without any change in their code.

Overview In section 2, we present the system model and definitions that will be used throughout the paper. In section 3, a distributed Eulerian traversal algorithm is presented, along with associated correctness and complexity results. Applications to self-stabilization and the cut-through routing are given in Section 4. Concluding remarks are provided in Section 5.

2 Model

A *processor* is a sequential deterministic machine that uses a local memory, a local algorithm and input/output capabilities. Such a processor executes its local algorithm, that modifies the state of the processor memory, and send/receive messages using the communication ports. An *unidirectional communication link* transmits messages from a processor o (for *origin*) to a processor d (for *destination*). The link is interacting with one input port of d and one output port of o. We assume that links do not loose, reorder or duplicate messages.

A *distributed system* is a 2-tuple $\mathcal{S} = (P, L)$ where P is the set of processors and L is the set of communication links. A distributed system is represented by a directed graph whose nodes denote processors and whose directed edges denote communication links. The state of a processor can be reduced to the state of its local memory, the state of a communication link can be reduced to its contents, then the global system state, called a *configuration*, is the product of the states of memories of processors of P and of contents of communication links in L. The set of configurations is denoted by $\mathcal{C}$.

Our system is not fixed once for all: it passes from a configuration to another when a processor executes an instruction of its local algorithm or when a communication link delivers a message to its destination. This sequence of reached configurations is called a *computation*, and is a maximal sequence of configurations of $\mathcal{S}$ denoted by $C_1, C_2, \ldots$ and such that for any positive integer i, the transition from C_i to C_{i+1} is done through execution of an atomic action of every element of a non empty subset of P and/or L. Configuration C_1 is called the *initial configuration* of the computation. In the most general case, the specification of a problem is by enumerating computations that satisfy this problem. Formally, a *specification* is a set of computations. A computation E satisfies a specification $\mathcal{A}$ if it belongs to $\mathcal{A}$.

A self-stabilizing algorithm does not always verify its specification. However, it seeks to reach a configuration from which any computation will verify its specification. A set of configurations $B \subset \mathcal{C}$ is *closed* if for any $b \in B$, any possible computation of system $\mathcal{S}$ whose b is initial configuration only contains configurations in B. A set of configurations $B_2 \subset \mathcal{C}$ is an *attractor* for a set of configurations $B_1 \subset \mathcal{C}$ if for any $b \in B_1$ and any possible computation of $\mathcal{S}$ whose initial configuration is b, the computation contains a configuration of B_2. Then a system $\mathcal{S}$ is *self-stabilizing* for a specification $\mathcal{A}$ if there exists a non-empty set of configurations $\mathcal{L} \subset \mathcal{C}$ such that (**closure**) any computation of $\mathcal{S}$ whose initial configuration is in $\mathcal{L}$ satisfies $\mathcal{A}$ and, (**convergence**) $\mathcal{L}$ is an attractor for $\mathcal{C}$.

In this paper, we also use a weaker requirement than self-stabilization that we call node-stabilization. Informally, a system is node-stabilizing if it reaches a correct behavior independently of the initial state of the nodes, yet one may assume that the state of the communications links satisfies some global predicate. Then a system $\mathcal{S}$ is *node-stabilizing* for a specification $\mathcal{A}$ if there exists two non-empty sets of configurations $\mathcal{L} \subset \mathcal{C}$ and $\mathcal{N} \subset \mathcal{C}$ such that (**closure**) any computation of $\mathcal{S}$ whose initial configuration is in $\mathcal{L}$ satisfies $\mathcal{A}$, (**convergence**) $\mathcal{L}$ is an attractor for $\mathcal{N}$, and (**node independence**) all possible node states are in $\mathcal{N}$.

3 State-optimal distributed Eulerian traversal

In this section, we present a distributed algorithm that stabilizes to an Eulerian traversal provided that it is executed starting from a configuration where a single message is present (either at a node of within a communication link). In the following, we call such a configuration a *singular configuration*.

While the time complexity of this algorithm is not optimal ($O(m^2)$ starting from the worst possible initial configuration, while [10] provides a $O(m)$ distributed algorithm, where m is the overall number of edges of the network), it does have nice static (it is state optimal at every node) and dynamic (it is node-stabilizing) properties.

3.1 The Algorithm

We assume $\delta^-(i)$ and $\delta^+(i)$ denote respectively the incoming and outgoing degree of node i. As the communication graph is Eulerian, let $d_i = \delta^-(i) = \delta^+(i)$. Moreover, each processor P_i has a $Path_i$

*(a)*The system at the beginning of first round *(b)*The system at the beginning of second round

*(c)*The system at the beginning of third round *(d)*The system at the beginning of fourth round

Figure 1: Example of computation of Algorithm 1

variable, that takes values between 0 and $d_i - 1$. All operations on this variable are done modulo d_i. Algorithm 1, that is executed upon receipt of a message m, is the same for all processors in the system.

Algorithm 1 *Distributed Eulerian traversal algorithm at node i*

```
    Send $m$ using the outgoing link whose index is $\mathrm{Path}_i$
    $\mathrm{Path}_i \leftarrow \mathrm{Path}_i + 1$
```

Example of computation Figure 1.a presents a distributed system whose communication graph is Eulerian: processors A, C, and F each have one incoming link and one outgoing link; processors B, D and E each have two incoming links and two outgoing links. The $Path_i$ variable of each processor P_i is denoted by an arrow that points to the outgoing link on which the next message will be sent to. For example, processor A has just sent message m and processor B (which is about to receive m) will retransmit it through its outgoing link b_2. We now follow the path of message m from "initiator" A (actually the latest processor that transmitted m). Given the initial $Path_i$ variables configuration, m will go through links a_1, b_2, d_1 and c_1 before it returns to processor A. The followed path is obviously not Eulerian, since links b_1, d_2, e_1, e_2 and f_1 have not been visited by m. Nevertheless, variables $Path_B$ and $Path_D$ have changed their values during this round of message m: $Path_B$ now points to b_1 and $Path_D$ to d_2.

Figure 1.b presents the same distributed system as Figure 1.a, but at the second round of message m. Given the configuration of the $Path_i$ variables, message m follows links a_1, b_1, e_1, d_2, b_2, d_1 then c_1 before returning to processor A. Again, the followed path is not Eulerian, since links e_2 and f_1 have not been followed by m. Yet, the $Path_E$ variable has changed value during this second round of m: it now point to e_2. At the contrary, variables $Path_B$ and $Path_D$ are back to the values they had at the beginning of second round (*i.e.* b_1 and d_2, respectively).

Figure 1.c presents the same system at the beginning of third round. Given the $Path_i$ variables configuration, the message will follow links a_1, b_1, e_2, f_1, e_1, d_2, b_2, d_1 then c_1 before returning to processor A. The followed path is Eulerian, since every link is traversed exactly once, and that message m is back to the "initiator" A. Moreover, $Path_i$ variables hold the same values as at the beginning of third round (see Figure 1.d), which means that the fourth round will be identical to the third. Consequently, message m will follow the very same Eulerian path infinitely often.

3.2 Proof of correctness

We wish to prove that Algorithm 1 is node-stabilizing for the Eulerian traversal problem. Each of the following lemmas assume that the algorithm is started from a singular configuration as defined below:

Definition 2 *A configuration C is* singular *if it contains exactly one message (either traversing a node or a communication link).*

The liveness lemma (Lemma 3) shows that Send actions through a particular link appear infinitely often in any computation, and so for any particular link. The following lemmas make use of it and consider computation factors that begin and end with the same Send action. The uniqueness lemma (Lemma 5) shows that between any two successive Send actions on the same link, no other link may be related to more than one Send action. The completeness lemma (Lemma 6) shows that after every link is related to at least one Send action, between any two successive Send action on a particular link, every other link is related to exactly one Send action. Finally, the legitimacy lemma (Lemma 7) shows that these Send actions appear always in the same order, and thus that the message performs an Eulerian traversal forever.

Lemma 3 *Starting from a singular configuration, every link is visited by the message infinitely often.*

Proof. Suppose there exists a link $c_{i \to j}$ (allowing P_i to send messages to P_j) that is not visited infinitely often starting from a singular configuration. From Algorithm 1, if processor P_i executes Send actions infinitely often, it does so on every outgoing link. Then, if P_i did not execute a Send action infinitely often on link $c_{i \to j}$, then P_i has executed Send action only a finite number of times in the whole computation, and thus P_i received the message only a finite number of times. Every incoming link P_i are then in the same case as $c_{i \to j}$, and have not been visited infinitely often. Applying the same reasoning again and since the network is finite and strongly connected, no link has been visited infinitely often. This contradicts the fact that Algorithm 1 may not deadlock starting from a singular configuration (since every receipt of a message implies an immediate Send action). ■

Notation 4 *For the sake of simplicity in the proof of the following lemmas, we arbitrarily number links from 1 to L (the number of links in the system) and denote by l_j a* Send *action through link number j. Thus l_j and l_p denote* Send *action on different links if $j \neq p$ and on the same link if $j = p$.*

Lemma 5 *Starting from a singular configuration, between any two* Send *actions on the same link, no other link is associated with a* Send *action twice.*

Proof. Let us consider a particular computation e of Algorithm 1 and its projection p on Send actions. From Lemma 3, action l_1 appears infinitely often in p. Let us study a factor f of p that starts and ends with l_1 and such that f does not contain any other l_1.

We do not discuss the trivial case $f = l_1 l_1$ where no other Send action that on the unique link is possible (in this case, the Eulerian traversal is trivially satisfied). Now assume that between the two l_1 actions, f contains some action l_k twice: $f = l_1 l_2 \dots l_k \dots l_n l_k \dots l_1$. In more details, while the n first actions of f are pairwise distinct, the second l_k is the first action that appears twice in f. We will show that the existence of l_k leads to a contradiction.

Action l_k is a Send action on link $c_{i \to j}$. If P_i performed twice a Send action involving $c_{i \to j}$ (the two occurrences of l_k), then P_i received $\delta^+(i)+1$ times the message. In turn, if P_i received $\delta^+(i)+1$ times the message, and since the graph is Eulerian, P_i received it $\delta^-(i)+1$ times and thus twice from the same incoming link. Then it follows that two Send actions occurred on this incoming link. In the writing of $f = l_1 l_2 \dots l_k \dots l_n l_k \dots l_1$, this means that some two actions between l_1 and l_n are identical, while our hypothesis claims that they are pairwise distinct. Therefore every factor f of p of length $n+1$ that starts and ends with l_1 can be written as $f = l_1 \dots l_p l_1$. ■

Lemma 6 *Starting from a singular configuration, and after every link has been visited by the message, between any two Send actions on the same link, every other link is associated with a Send action exactly once.*

Proof. Let us consider a computation of the algorithm. From Lemma 3, this computation contains each Send action infinitely often. It is then possible to write its projection p on Send actions as: $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$ where t_0 contains at least once each of the $l_{k \in \{1,\dots,L\}}$ and where none of the $t_{i \geq 1}$ contains l_1. From Lemma 5, it is impossible that any of the $t_{i \geq 1}$ contains the same Send action $l_{k \neq 1}$ twice. Therefore, all factors $t_{i \geq 1}$ are of length at most $L-1$.

Suppose now that the factor t_j ($j \geq 1$) is of length strictly lower than $L-1$ and let us denote by l_p ($p \neq 1$) the Send action that does not appear in t_j. Lemma 3 ensures that l_p appears infinitely often in the computation. Thus there exists a smallest $k > j$ such that l_p is a send action of factor t_k. Moreover, since l_p appears by definition in t_0, there also exists a greatest $m < j$ such that l_p is a Send action of factor t_m.

Consequently, the projection p has a factor $t_m l_1 t_{m+1} \dots t_j \dots t_{k-1} l_1 t_k$ where l_p ($p \neq 1$) does not appear in any of the $t_{i \in \{m+1,\dots,k-1\}}$ but appears in t_m and in t_k:

$$\underbrace{\dots l_p \dots}_{t_m} \overbrace{l_1 t_{m+1} \dots t_j \dots t_{k-1} l_1}^{\text{no } l_p} \underbrace{\dots l_p \dots}_{t_k}$$

The Send action l_1 then appears twice between two successive Send actions l_p, which contradicts Lemma 5.

In conclusion, in the projection factor $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$, every $t_{i \geq 1}$ contains only different Send actions and is of size $L-1$. In other terms, after every link has been visited by the message (*i.e.* after t_0), between any two Send actions on the same link l_1, every other link is associated with a Send action exactly once (every $t_{i \geq 1}$ contains each $l_{k \in \{2,\dots,L\}}$ exactly once). ■

Lemma 7 *Starting from a singular configuration, and after every link has been visited by the message, between any two Send actions on the same link, every other link is associated with a Send action exactly once and in the same order.*

Proof. Let us consider a computation of the algorithm. From Lemma 3, this computation contains each Send action infinitely often. It is then possible to write its projection p on Send actions as: $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$ where t_0 contains at least once each of the $l_{k \in \{1,\dots,L\}}$ and where (from Lemma 6) every $t_{i \geq 1}$ contains exactly once each of the $l_{k \in \{2,\dots,L\}}$. In more details, for $t_j = l_2 l_3 \dots l_L$ we can write t_{j+1} as $l_{\sigma(2)} l_{\sigma(3)} \dots l_{\sigma(L)}$, where σ is a permutation. Assume that there exists a smallest integer $q_1$in $\{2,\dots,L\}$ and such that $\sigma(q_1) \neq q_1$. Then there exists an integer q_2 ($q_1 < q_2 \leq L$) and such that $\sigma(q_1) = q_2$.

We are now able to rewrite a factor of the projection p as:

$$\underbrace{l_2 l_3 \dots l_{q_1-1} l_{q_1} l_{q_1+1} \dots l_{q_2-1} l_{q_2} \dots l_L}_{t_j} \, l_1 \, \underbrace{l_2 l_3 \dots l_{q_1-1} l_{q_2} l_{\sigma(q_1+1)} \dots l_{\sigma(q_2-1)} l_{q_1} \dots l_{\sigma(L)}}_{t_{j+1}} \, l_1$$

Then, between two l_{q_1}, Lemma 5 is contradicted. Indeed, between two successive occurrences of l_{q_1}, we find two occurrences of l_{q_2}. This contradiction permits to prove that every possible permutation σ is reduced to the identity and that the projection p can be written as $t_0(l_1 t_1)^\omega$. After every link has been visited by the message head (after t_0), between any two Send actions on the same link l_1, every other link is associated with a Send action exactly once and in the same order as in t_1. ■

Theorem 8 *Starting from an singular configuration, Algorithm 1 stabilizes to an Eulerian traversal.*

Proof. In Lemma 7, we proved that any computation has a factor of the projection p on Send actions of the form $t_0(l_1 l_2 \dots l_L)^\omega$, where t_0 is finite. Consequently, an Eulerian traversal through links 1 to L is performed infinitely often after a finite number of message exchanges. ■

3.3 Complexity

In order to know the outgoing link to which a message is to be sent, a processor P_i requires d_i states. Similarly, to know the incoming link by which a message was receipt, P_i requires d_i states. We show that a d_i states memory per processor is necessary for distributed Eulerian traversal.

Lemma 9 *Every distributed Eulerian traversal algorithm requires d_i states at processor P_i.*

Proof. Suppose that there exists an Eulerian traversal algorithm (deterministic or probabilistic, stabilizing or non-stabilizing) such that there exists at least one processor P_i that uses at most $d_i - 1$ states. In an Eulerian network, every processor P_i has at least one incoming and one outgoing edges. Thus for any P_i, $d_i \geq 1$. If $d_i = 1$, then if P_i has less than one state, it may execute no code. Now, if $d_i \geq 2$, assume that P_i has at most $d_i - 1$ states.

In this last case, there exists at least two incoming links c_1 and c_2 of P_i by which P_i received the message and such that P_i had to perform a Send action using the same outgoing link c_3 twice (in case P_i's algorithm is deterministic) or twice with probability $\epsilon > 0$ (in case P_i's algorithm is probabilistic). Then the message forwarding scheme is not Eulerian, since at any point in the computation, it is either certain or possible that two incoming links are not forwarded to two different outgoing links. ■

A direct corollary of this lemma is the following theorem.

Theorem 10 *Algorithm 1 is state optimal.*

For the time complexity part, a direct consequence of Lemma 5 is the following theorem.

Theorem 11 *Algorithm 1 performs its first traversal whithin $O(m^2)$ message exchanges.*

4 Applications

In this section, we investigate applications of Algorithm 1 in the context of self-stabilization. Strictly speaking, Algorithm 1 is not self-stabilizing, since its correct behavior requires that it is started from a singular configuration (a configuration where a single message is present). However, it does stabilizes to an Eulerian traversal independently of nodes initial state and messages actual contents. Randomization enables to make Algorithm 1 self-stabilizing without the overkill of using a self-stabilizing mutual exclusion algorithm to guarantee uniqueness of the message; then the resulting self-stabilizing virtual circuit construction algorithm shows some interest particularly in the context of cut-through routing, where nodes must retransmit messages before they are done receiving them. Due to space constraints, all proofs in this section are only informally sketched, yet the interested reader may refer to [13].

4.1 Reaching a singular configuration

Since [9] showed that message-passing self-stabilizing algorithms require timeouts to handle the case where no message is initially present in the network, we concentrate here on eliminating superfluous messages in the case where the number of initially present messages is greater or equal to 2. Our solution is by giving different randomized speeds to messages so that in an infinite computation, it is possible that two messages are present at the same node at the same time. The result of that event is the node discarding every message but one.

Multiple speeds If the system is asynchronous (the communication time between the origin and the destination of a link may be arbitrary), we assume a random distribution on communication time, so that the speeds of the messages are actually different.

If the system is synchronous (the communication time between the origin and the destination of a link is bounded by 1), we can split every computation in *global steps* during which every message is sent and received, and assume that nodes dispose of a random Boolean variable. At each global step, every node that receives a message consults its random variable: if the random variable returns *true*, then the node holds the message one more global step; otherwise it sends the message immediately. Note that a node may hold a message for at most one global step, and that the induced relative speeds on messages are now different and randomized. This technique has a low memory footprint since a node needs only to know if it should wait one more global step (one bit is sufficient).

Message decreasing Now assuming that messages do have different speeds, we show that Algorithm 1 stabilizes to a single message configuration. Indeed, starting from an arbitrary configuration with at least two messages, two cases may appear:

1. At least two messages follow the same circuit that goes through all links in the Eulerian graph. By the probabilistic setting, these two messages have different speeds and thus starting from any configuration, there is a positive probability that they are at the same node, which will discard all messages but one, so that the overall number of messages decreases.
2. At least two messages follow two different circuits, but from the rotating exploration nature of Algorithm 1, these two circuits do share a common node i. Then, from the probabilistic speeds of these two messages, starting from any configuration, there is a positive probability that at some point in the computation, they are at the same node, which will discard all messages but one. Then the overall number of messages decreases.

Since at any time there is a positive probability that, in a finite number of steps, the number of messages decreases if it is strictly greater that 1, then by the main theorem of [3], after finite time a single message remains in the system with probability 1.

4.2 Towards a virtual ring construction

Since self-stabilization was first presented by Dijkstra in 1974 (see [6]), which provided three mutual exclusion algorithm on unidirectional ring networks, numerous works in self-stabilization were proposed on unidirectional rings (see [11] or [7]). Therefore, it is interesting to provide a scheme that permits to run such algorithms on more general networks, by constructing a *virtual ring* topology on top of which the original algorithm is run.

Many self-stabilizing solutions to the virtual ring construction problem exists for bidirectional networks (which are a proper subset of Eulerian networks); many of these works first construct a spanning tree of the graph, then perform an Eulerian tour of the spanning tree (see [7]). In general strongly connected networks (which are a proper superset of Eulerian networks), approaches by Tchuente ([12]) and Alstein *et al.* ([1]) make use of a central preprocessing of the communication graph, or assume that nodes are given unique identifiers. Two drawbacks of [1, 12] is the high memory consumption and the fact that nodes simulate several processes in the virtual ring, which may be incorrect for some applications (such as [2]). In addition, and to the best of our knowledge, none of the aforementioned approaches can be used in non-ring networks when the cut-though routing scheme is used.

Our solution circumvents many of the previously mentioned drawbacks: the class of Eulerian graphs that we consider is intermediate between bidirectional and strongly connected graphs classes, we do not require central preprocessing nor unique node identifiers, memory consumption is low ($O(d)$ at each node, where d is the outgoing degree of the node), and efficient cut-through routing is supported.

Cut-through routing compliance The two main reasons for the cut-through routing compliance are the following: *(i)* since the underlying graph is Eulerian, each node has as many incoming links as outgoing links, so when a message arrives, it may be forwarded immediately to a free outgoing link, and *(ii)* since the message contents is unused in the forwarding scheme, no additional processing is needed before giving control to the composed cut-through algorithm (that could be any of [2, 4, 14]).

Virtual circuit bijection In addition, the Eulerian property of the traversal guarantees that each link is visited exactly once at each traversal. Thus, if a node has d outgoing links, then the link that is locally labeled 0 at this node is visited exactly once at each Eulerian traversal, no matter how the local labeling on outgoing links is performed. Assume now that our Eulerian traversal algorithm is run to build a virtual circuit that is used by an upper layer application (such as [2]). If the upper layer application is activated only when a message arrives and the $Path_i$ variable equals 0, then we are guaranteed that this upper application is activated exactly once at each Eulerian traversal. This means that at the upper application level, there is a bijection between the nodes in the actual system and the nodes in the virtual ring system. This bijection is usually required for sake of correctness or service time guarantee.

5 Concluding remarks

We presented a state-optimal distributed solution to the Eulerian traversal problem. Each node only needs d memory states, where d is the node outgoing degree. Our algorithm also presents some failure resilience properties: it is independent of the message contents and after $O(m^2)$ message exchanges, it stabilizes to an infinite Eulerian traversal whatever the initial configuration of the nodes may be (it is node-stabilizing).

The message content independence was shown useful in the context of cut-through routing, since a node need not know the contents of a message to properly route it. The insensitivity to node initialization was extended by means of randomization so that the resulting system is self-stabilizing. This solution permits to avoid high memory consumption and preprocessing that were required by previous approaches.

References

[1] D. Alstein, J. H. Hoepman, B. E. Olivier , and P.I.A. van der Put. Self-stabilizing mutual exclusion on directed graphs. Technical Report CS-R9513, CWI, 1994. Published in *Computer Science in the Netherlands* (CSN 94), pp. 45–53.

[2] J. Beauquier, A. K. Datta, and S. Tixeuil. Self-stabilizing Census with Cut-through Constraints. In Proceedings of the *Fourth Workshop on Self-stabilizing Systems* (WSS'99), Austin, Texas. pp. 70-77, May 1999.

[3] J. Beauquier, M. Gradinariu, and C. Johnen. In Proceedings of the International Conference on *Principles of Distributed Computing* (PODC'99), Atlanta, pp. 199-208, 1999.

[4] A. M. Costello and G. Varghese. The FDDI MAC meets self-stabilization. In Proceedings of the *Fourth Workshop on Self-stabilizing Systems (WSS'99)*, Austin, Texas. pp. 1-9, May 1999.

[5] X. Deng and C. H. Papadimitriou. Exploring an unkown graph. In Proceedings of the 31th Annual IEEE Symposium on *Foundations of Computer Science*, Vol. I, pp. 355-361, 1990.

[6] E. W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643-644, 1974.

[7] S. Dolev. Self-stabilization. *The MIT Press.* 2000.

[8] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911-921, 1991.

[9] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448-458, 1991.

[10] M. Hadim M. and A. Bouabdallah. A distributed algorithm for constructing an Eulerian cycle in networks. In Proceedings of *International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'99), Las Vegas, USA, June 28th-July 1st, 1999.

[11] T. Herman. A Comprehensive Bibliography on Self-Stabilization. A Working Paper in the *Chicago Journal of Theoretical Computer Science*. Available at `http://www.cs.uiowa.edu/ftp/selfstab/bibliography/`.

[12] M. Tchuente. Sur l'auto-stabilisation dans un réseau d'ordinateurs. *RAIRO Informatique Théorique*, 15:47-66, 1981.

[13] S. Tixeuil. Auto-stabilisation efficace. *Ph.D. Thesis, Université de Paris Sud*, France. Jan. 2000. Available at `http://www.lri.fr/~tixeuil`.

[14] S. Tixeuil and J. Beauquier. Self-stabilizing Token Ring. In Proceedings of *International Conference on System Engineering* (ICSE'96), Las Vegas, Nevada. Jul. 1996.