Tolerating Transient and Intermittent Failures[1]

Sylvie Delaët

and

Sébastien Tixeuil
Laboratoire de Recherche en Informatique, UMR CNRS 8623,
Université de Paris Sud, 91405 Orsay Cedex, France
E-mail: `delaet@lri.fr`, `tixeuil@lri.fr`

Version: January 18, 2002

Fault tolerance is a crucial property for recent distributed systems. We propose an algorithm that solves the census problem (list all processor identifiers and their relative distance) on an arbitrary strongly connected network.

This algorithm tolerates transient faults that corrupt the processors and communication links memory (it is self-stabilizing) as well as intermittent faults (fair loss, reorder, finite duplication of messages) on communication media. A formal proof establishes its correctness for the considered problem. Our algorithm leads to the construction of algorithms for any silent problems that are self-stabilizing while supporting the same communication hazards.

Key Words: **self-stabilization, unreliable communication, census**

1. INTRODUCTION

We consider distributed systems that consist of a collection of processors linked to each other by communication media that allow them to exchange messages. As larger systems imply less reliability, several kinds of failures can be classified according to their locality (processor or link) and their nature.

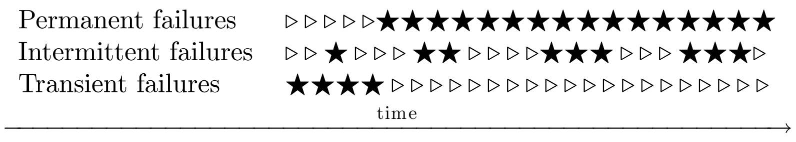
1. *Permanent failures* are failures that make one or several components of the system stop running forever. In the case of a processor, it stops executing its program forever. In the case of a communication link, this can be interpreted as a definitive rupture of the communication service.

2. *Intermittent failures* are failures that make one or several components of the system behave erratically from time to time. A processor can have a Byzantine behavior, and a communication link may loose, duplicate, reorder or modify messages in transit.

3. *Transient failures* are failures that place one or several components of the system in some arbitrary state, but stop occurring after some time. For a processor, such failures may occur following a crash and repair operation, or temporary shutdown of its power supply. For a communication link, electromagnetic fields may lead to similar problems.

[1]An extended abstract of a preliminary version of this paper appeared in [13]. This work was supported in part by the french STAR project.

The time organization of permanent, intermittent and transient failures is presented below, where ▷ denotes a correct behavior of the system, and where ★ denotes a failure of the system.



1.1. Failure-tolerance in Distributed systems

Robustness is one of the most important requirements of modern distributed systems. Two approaches are possible to achieve fault-tolerance: on the one hand, robust systems use redundancy to mask the effect of faults, on the other hand, self-stabilizing systems may temporarily exhibit an abnormal behavior, but must recover correct behavior within finite time.

Robustness When a small number of the system components fail frequently (this is the case for permanent and intermittent failures), robust systems should always guarantee a correct behavior. Most often, such approaches make the hypothesis of a limited number of faults, due to impossibility results even when communication links are reliable. A fundamental result (see [20]) shows that the Consensus problem (all processors agree on a common initial value) is impossible to solve deterministically in an asynchronous system even if the failure is permanent and limited to a single processor. In the case of intermittent failures on communication links, most works deal with transformation of unreliable links into reliable links (for higher level messages). The case of fair loss was solved in [6, 27], but such constructions are impossible when links may duplicate (finitely) and reorder messages (see [28]).

Self-stabilization Conversely, when all system components behave correctly most of the time (this is the case with transient failures), one could accept temporary abnormal service when that system suffers from a general failure, as long as recovery to a correct behavior is guaranteed after finite time. This approach, called self-stabilization, consists in always behaving as if all system components were correct. At the contrary of fault-tolerance, self-stabilization does not make any restriction of the subset of the system that is hit by the failure. Since its introduction by Dijkstra (see [14]), a growing number of self-stabilizing algorithms solving different problems have been presented (see [16]). In particular, several recent publications prove that being able to start from any arbitrary configuration is desirable as a property of fault tolerance. For example, [23] shows that processor crashes and restarts may lead a system to an arbitrary global state, from which a self-stabilizing algorithm is able to recover.

1.2. Related Work

Historically, research in self-stabilization over general networks has mostly covered undirected networks where bidirectional communication is feasible (the Update protocol of [18], or the algorithms presented in [3, 19]). For example, the self-stabilizing algorithms that are built upon the paradigm of local checking (see

[5]) use bidirectional communication to compare one node state with those of its neighbors and check for consistency. The lack of bidirectional communication was overcome in recent papers using strong connectivity (which is a weaker requirement than bidirectional) to build a virtual well known topology on which the self-stabilizing algorithm may be run (a tree in [1]). As many self-stabilizing algorithms exist for rings or trees in the literature, these constructions may be used to reuse existing algorithms in general networks.

Several algorithms are self-stabilizing and tolerate a limited amount of processor crash failures (see [4, 21, 9]). However, they are studied in a communication model that is almost reliable (links are only subject to transient failures). Most related to our problem is [25], where Masuzawa presents a self-stabilizing Topology Update algorithm that also support, to a certain extent, permanent processor failures. In [7], the authors consider the case of systems subject to crash failures for processors and intermittent failures for links (only the loss case is considered). However, in their approach, bidirectional communication link are assumed to provide a lower level communication protocol that is reliable enough for their purpose. To some extent, topology changes can be considered as permanent failures on links. In this context, Super-stabilizing and Snap-stabilizing protocols (introduced in [18] and [10], respectively) are self-stabilizing protocols that also tolerate limited topology changes. In [2], Afek and Brown consider self-stabilizing algorithms along with lossy communication links, but they assume bidirectional communications in order to build an underlying self-stabilizing data-link protocol. Finally, [22] consider the construction of wait-free objects in a self-stabilizing setting.

1.3. The Census Problem

The Census problem is derived from the Topology Update task by removing the location information requirement. Informally, a self-stabilizing distributed algorithm that solves the Census problem must ensure that eventually, the system reaches a global state where each processor knows the identifiers of all processors in the network and their relative distances to itself. Census information is sufficient to solve many fundamental tasks, such as leader election (the processor with minimum identifier is elected), counting all nodes in the system (the number of processors in the Census list), or topological sort of ancestors. Typically, a Topology Update algorithm would require each processor in the system to store each link of the communication graph (inducing a $\Omega(N^2)$ bits storage at each node, where N is the size of the network), while a Census algorithm would require each processor in the system to store each node of the communication graph along with its relative distance (inducing a $\Omega(N \times \log_2 N)$ bits storage at each node).

A self-stabilizing solution (although not presented as such) to the problem of Topology Update has been proposed in [26]. Subsequent works on self-stabilization and the Topology Update problem include [25, 15, 18], but none of the aforementioned protocols consider intermittent link failures. Those algorithm typically use $O(\delta \times N^2)$ bits per node when links can be enumerated, where δ is the degree of the node, and $O(\delta \times N^2 \times \log_2(k))$ bits otherwise, where k is the number of possible identifiers for nodes. These works share two common points: *(i)* communication between neighboring nodes is carried out using reliable bidirectional links, and *(ii)* node are aware of their local topology. In the context of *directed* networks, both points *(i)* and *(ii)* are questionable:

1. If a bidirectional network is not available, then self-stabilizing data link proto-

cols (that are acknowledgment based, such as those presented in [2]) can not be used to transform any of those works so that they perform in unreliable message passing environments.

2. In directed networks, it is generally easy to maintain the set of input neighbors (by checking who has "recently" sent a message), but it is very difficult (if not impossible) to maintain the set of output neighbors (in a satellite network, a transmitter is generally not aware of who is listening to the information it communicates).

Two algorithms [12, 8] were previously presented for the Census problem in unidirectional networks. They are both self-stabilizing and assume the simple topology of a unidirectional ring: [8] assumes reliable communications and supports the efficient cut-through routing scheme, while [12] supports fair loss, finite duplication, and desequencing of messages.

1.4. Our Contribution

We extend the result of [12] from unidirectional rings, which are a small subset of actual communication networks, to general strongly connected networks, that are a proper superset of unidirectional rings and bidirectional networks. However, we retain all link failure hypothesis of [12]: fair loss, finite duplication, reordering. In more details, we present a self-stabilizing algorithm for the Census problem that tolerates message loss, duplication and reordering both in the stabilizing phase and in the stabilized phase. Our algorithm only assume that the input neighborhood of each node is known, but not the output neighborhood, so that it can be used in a large class of actual systems. Our algorithm requires $O(N \times (\log_2(k) + \delta_i^-))$ bits per node, where k is the number of possible identifiers for nodes, and where δ_i^- is the input degree of node i. The stabilization time is $O(D)$, where D is the diameter of the network.

Using the scheme presented in [18] on top of our algorithm, we are then able to solve any global computation task (*i.e.* any task that can be solved by a silent system) in a self-stabilizing way, and still cope with unreliable communication links even when stabilized.

Although we assume the system communication graph is strongly connected, we do not use this information to build a well known topology (*e.g.* a ring) and run a well known algorithm on it. Indeed, this approach could potentially lower the performance of the overall algorithm, due to the fact that the communication possibilities are not used to their full extent. As a matter of fact, when our distributed algorithm is run on a network that is not strongly connected, we ensure that the collected information at each node is a topologically sorted list of its ancestors. In DAG (directed acyclic graph) structured networks, such kind of information is often wished (see [11]), and our approach makes it tolerant to link failures for free.

1.5. Outline

Section 2 presents a model for distributed systems we consider, as well as convenient notations used in the rest of the paper. Section 3 describes our self-stabilizing census algorithm on strongly connected networks, while Section 4 establishes its proof of correctness. Concluding remarks are proposed in Section 5.

2. PRELIMINARIES

2.1. Distributed Systems

In order to present a formal proof of our algorithm, we introduce a few definitions and notations that describe the model used in the rest of the paper.

A *processor* is a sequential deterministic machine that uses a local memory, a local algorithm and input/output capabilities. Intuitively, such a processor executes its local algorithm. This algorithm 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. Depending on the way messages are handled by a communication link, several properties can be defined on a link. [24] proposes a complete formalization of these properties. We only enumerate those that are related to our algorithm. There is a *fair loss* when, infinitely messages being emitted by o, infinitely messages are received by d. There is *finite duplication* when every message emitted by o may be received by d a finite number of times: however, a bound on the number of time a message was duplicated is *not* known to the processors. There is *reordering* when messages emitted by o may be received by d in a different order than that they were emitted. We also assume that any message that is not lost is eventually received by d. In particular, if the origin node o *continuously* send the same message infinitely, then this message is eventually received by the destination node d.

A *distributed system* is a 2-uple $\mathcal{S} = (P, L)$ where P is a set of processors and L is a set of communication links. A distributed system is represented by a directed graph whose nodes denote processors and whose directed edges (or arcs) denote communication channels (or 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 can be simply defined as:

Definition 1. A *configuration* of a distributed system $\mathcal{S} = (P, L)$ 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 noted $\mathcal{C}$.

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*.

Definition 2. A *computation* of $\mathcal{S} = (P, L)$ is a maximal sequence of configurations of $\mathcal{S}$ noted $C_1, C_2, \ldots$ 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 solve this problem. In the special case of the Census problem, where a global deterministic calculus is done, the specification can be given in terms of a set of system configurations.

Definition 3. A configuration c satisfies the Census specification if and only if for any i in P, i knows the name and distance of all other elements relatively to itself in c.

A computation e satisfies the Census specification if and only if every configuration c in e satisfies the Census 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.

DEFINITION 4. 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 $\mathcal{A}$,

Convergence $\mathcal{L}$ is an attractor for $\mathcal{C}$.

To show that a given system is self-stabilizing, it is sufficient to exhibit a particular non-empty subset of configurations of the system : the *legitimate configurations*. Then it is to be shown that any computation starting from a legitimate configuration satisfies the considered problem (closure property), and that from any possible configuration of the system, any possible computation of the system leads to a legitimate configuration (convergence property).

2.2. System Settings

Constants Each node knows its unique identifier, which is placed in non corruptible memory. We denote this identifier as an italics Latin letter. Each node i is aware of its input degree δ_i^- (the number of its incident arcs), which is also placed in non corruptible memory. A node i arbitrarily numbers its incident arcs using the first δ_i^- natural numbers. When receiving a message, the node i knows the number of the corresponding incoming link (that varies from 1 to δ_i^-).

Local memory Each node maintains a local memory. The local memory of i is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$. Each i_α is a non-empty list of pairs $\langle$*identifier*,*colors*$\rangle$, where *identifier* is a node identifier, and where *colors* is an array of booleans of size δ_i^-. Each boolean in the *colors* array is either *true* (denoted by $\bullet$) or *false* (denoted by $\circ$). We assume that natural operations on boolean arrays, such as unary *not* (denoted by $\neg$), binary *and* (denoted by $\wedge$) and binary *or* (denoted by $\vee$) are available.

The goal of the Census algorithm is to guarantee that the local memory of each node contains the list of lists of identifiers (whatever the *colors* value in each pair $\langle$*identifier*,*colors*$\rangle$) that are predecessors of i in the communication graph. For the Census task to be satisfied, we must ensure that the local memory of each node i can contain as many lists of pairs as necessary. We assume that a minimum of

$$(N-1) \times \left(\log_2(k) + \delta_i^-\right)$$

bits space is available at each node i, where N is the number of nodes in the system and k is the number of possible identifiers in the system (see Lemma 4).

For example,

$$((j, [\bullet \circ \circ]; q, [\circ \bullet \circ]; t, [\circ \circ \bullet])(z, [\bullet \bullet \bullet]))$$

is a possible local memory for node i, assuming that δ_i^- equals 3. From the local memory of node i, it is possible to deduce the knowledge that node i has about its ancestors. With the previous example, node j is a direct ancestor of i (it is in the first list of the local memory of i) and this information was carried through incoming channel number 1 (only the first position of the *colors* array relative to node j is *true*). Similarly, nodes q and t are direct ancestors of i and this information was obtained through incoming links 2 and 3, respectively. Then, node z is at distance 2 from i, and this information was received through incoming links numbered 1, 2, and 3.

Messages Each node sends and receives messages. The contents of a message is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$. Each i_α is a non-empty list of *identifiers*.

For example,

$$((i)(j;q;t)(z))$$

is a possible contents of a message. It means that i sent the message (since it appears first in the message), that i believes that j, q, and t are the direct ancestors of i, and that z is an ancestor at distance 2 of i.

Notations The *distance* from i to j is denoted by $d(i,j)$, which is the minimal number of arcs from i to j. We assume that the graph is *strongly connected*, so the distance from i to j is always defined. Yet, since the graph may not be bidirectional, $d(i,j)$ may be different from $d(j,i)$. The *age* of i, denoted by χ_i, is the greatest distance $d(j,i)$ for any j in the graph. The network *diameter* is then equal to

$$\max_i \chi_i = D$$

3. SELF-STABILIZING CENSUS

3.1. Overview

Our algorithm can be seen as a knowledge collector on the network. The local memory of a node then represents the current knowledge of this node about the whole network. The only certain knowledge a node may have about the network is local: its identifier, its incoming degree, the respective numbers of its incoming channels. This is the only information that is stored in non-corruptible memory.

The algorithm for each node consists in updating in a coherent way (according to its constant information, see Section 2.2) its knowledge upon receipt of other processors' messages, and communicating this knowledge to other processors after adding its constant information about the network. More precisely, each information placed in a local memory is related to the local name of the incoming channel that transmitted this information. For example, node i would only emit messages starting with singleton list (i) and then not containing i since it is trivially an ancestor of i at distance 0. Coherent update consists in three kinds of actions: the first two being trivial coherence checks on messages and local memory, respectively.

Check Message Coherence Since all nodes have the same behavior, when a node receives a message that does not start with a singleton list, the message is trivially considered as erroneous and is ignored. For example, messages of the form $(\underline{(j;q;t)}(k)(m;y)(p;z))$ are ignored.

Check Local Coherence Regularly and at each message receiving, a node checks for local coherence. We only check here for trivial inconsistencies (see the *problem*() helper function): a node is incoherent if there exist at least one pair $\langle \textit{identifier}, \textit{colors} \rangle$ such that *colors*$=$ $[\circ \cdots \circ]$ (which means that some information in the local memory was not obtained from any of the input channels). If a problem is detected upon time-out, then the local memory is reinitialized, else if a problem is detected upon a message receipt, the local memory is completely replaced by the information contained in the message.

Trust Most Recent Information When a node receives a message through an incoming channel, this message should contain an information that was constructed after its own and then more reliable. The node removes all previous information obtained through this channel from its local memory. Then it integrates new information and only keeps old information (from its other incoming channels) that does not clash with new information.

Example Assume that a message $\textit{mess} = ((j)(k;\ l)(m)(p;\ q\,; r;\ i))$ is received by node i through its incoming link 1 and that $\delta_i^- = 2$. The following informations can be deduced:

1. j is a direct ancestor of i (it appears first in the message),
2. k and l are ancestors at distance 2 of i and may transmit messages through node j,
3. m is an ancestor at distance 3 of i,
4. p, q and r are ancestors at distance 4 of i, j obtained this information through m.

These informations are compatible with a local memory of i such as:

$$((j,[\bullet\circ];q,[\circ\bullet])(\,k,[\bullet\circ];\ e,[\circ\bullet];\ w,[\circ\bullet])(m,[\circ\bullet];y,[\bullet\bullet])(p,[\bullet\circ];z,[\circ\bullet];h,[\bullet\circ]))$$

Upon receipt of message *mess* at i, the following operations take place: *(i)* the local memory of i is cleared from previous information coming from link 1, *(ii)* the incoming message is "colored" by the number of the link (here each identifier α in the message becomes a pair $\alpha,[\bullet\circ]$ since it is received by link number 1 and *not* by link number 2), and *(iii)* the local memory is enriched as in the following (where "$\leftarrow$" denotes information that was acquired upon receipt of a message, and where "$\rightarrow$" denotes information that is to be forwarded to the node output links):

$$\begin{array}{cccccc}
(& (q,[\circ\bullet]) & (e,[\circ\bullet];w,[\circ\bullet]) & (m,[\circ\bullet];y,[\circ\bullet]) & (z,[\circ\bullet]) &) \\
\leftarrow (& (j,[\bullet\circ]) & (k,[\bullet\circ];l,[\bullet\circ]) & (m,[\bullet\circ]) & \left(\begin{array}{c} p,[\bullet\circ];q,[\bullet\circ]; \\ r,[\bullet\circ];i,[\bullet\circ] \end{array}\right) &) \\
\\
\rightarrow (& (j,[\bullet\circ];q,[\circ\bullet]) & \left(\begin{array}{c} k,[\bullet\circ];e,[\circ\bullet]; \\ w,[\circ\bullet];l,[\bullet\circ] \end{array}\right) & (m,[\bullet\bullet];y,[\circ\bullet]) & \left(\begin{array}{c} p,[\bullet\circ];z,[\circ\bullet]; \\ q,[\bullet\circ];r,[\bullet\circ] \end{array}\right) &)
\end{array}$$

This message enabled the modification of the local memory of node i in the following way: l is a new ancestor at distance 2. This was acquired through incoming link number 2 (thus through node j). Nodes m and y are confirmed to be ancestors at distance 3, but *mess* sends information *via* nodes j and q, while y only transmits its informations *via* node q. Moreover, q and r are part of the new knowledge of ancestors at distance 4. Finally, although i had information about h $(h, [\bullet\circ])$ before receiving *mess*, it knows now that the information about h is obsolete.

3.2. Communication issues

The property of resilience to intermittent link failures of our algorithm is mainly due to the fact that each message is self-contained and independently moves towards a complete correct knowledge about the network. More specifically:

1. The fair loss of messages is compensated by the infinite spontaneous retransmission by each processor of their current knowledge.
2. The finite duplication tolerance is due to the fact that our algorithm is *idempotent* in the following sense: if a processor receives the same message twice from the same incoming link, the second message does not modify the knowledge of the node.
3. The desequencing can be considered as a change in the relative speeds of two messages towards a complete knowledge about the network. Each message independently gets more accurate and complete, so that their relative order is insignificant. A formal treatment of this last and most important part can be found in Section 4.

3.3. Helper Functions

We now describe helper functions that will enhance readability of our algorithm. Those functions operate on lists, integers and pairs $\langle$*identifier*,*colors*$\rangle$. The specifications of those functions use the following notations: l denotes a list of identifiers, p denotes an integer, lc denotes a list of pair $\langle$*identifier*,*colors*$\rangle$, Ll denotes a list of lists of identifiers, and Llc denotes a list of lists of pairs $\langle$*identifier*,*colors*$\rangle$.

We assume that classical operations on generic lists are available: $\setminus$ denotes the binary operator "minus" (and returns the first list from which the elements of the second have been removed), $\cup$ denotes the binary operator "union" (and returns the list without duplicates of elements of both lists), $+$ denotes the binary operator "concatenate" (and returns the list resulting from concatenation of both lists), $\sharp$ denotes the unary operator that returns the number of elements contained in the list, and [] takes an integer parameter p so that $l[p]$ returns a reference to the p^{th} element of the list l if $p \leq \sharp l$ (in order that it can be used on the left part of an assignment operator ":="), or expand l with $p - \sharp l$ empty lists and returns a reference to the p^{th} element of the updated list if $p > \sharp l$.

clean$(lc, p)\rightarrow$ **list of couples** returns the empty list if lc is empty and a list of pairs lc_2 such that for each $\langle identifier_{lc},\ colors_{lc}\rangle \in lc$, if $colors_{lc} \wedge \neg colors(p) \neq [\circ \cdots \circ]$, then $\langle identifier_{lc},\ colors_{lc} \wedge \neg colors(p)\rangle$ is in lc_2, else $\langle identifier_{lc},\ *\rangle$ is not in lc_2.

Example: assuming $\delta_i^- = 3$,

$clean((j, [\circ\bullet\circ]; q, [\circ\circ\bullet]; k, [\circ\bullet\circ]; e, [\circ\circ\bullet]; w, [\circ\circ\bullet]; m, [\circ\circ\bullet]; y, [\circ\bullet\bullet]; p, [\circ\bullet\circ]; z, [\circ\circ\bullet]; h, [\circ\bullet\circ]), 2)$

$= (q, [\circ\circ\bullet]; e, [\circ\circ\bullet]; w, [\circ\circ\bullet]; m, [\circ\circ\bullet]; y, [\circ\circ\bullet]; z, [\circ\circ\bullet])$

colors(p) $\rightarrow$ **array of booleans** returns the array of booleans that correspond to the p^{th} incoming link, *i.e.* the array that is such that $[\ \underbrace{\circ \cdots \circ}_{p-1 \text{ times}}\ \bullet\ \underbrace{\circ \cdots \circ}_{\delta_i^- - p \text{ times}}\ ]$.

Example: assuming $\delta_i^- = 3$, $colors(2) = [\circ \bullet \circ]$

emit(i, Llc) sends the message resulting from (i)+$identifiers(Llc)$ on every outgoing link of i.

identifiers(Llc) $\rightarrow$**list of list of identifiers** returns the empty list if Llc is empty and returns a list Ll of list of identifiers (such that each pair $\langle identifier, colors\rangle$ in Llc becomes $identifier$ in Ll) otherwise.

Example: assuming $\delta_i^- = 3$,

$identifiers((j, [\circ\bullet\circ]; q, [\circ\circ\bullet])(k, [\circ\bullet\circ]; e, [\circ\circ\bullet]; w, [\circ\circ\bullet])(q, [\circ\bullet\circ])(k, [\circ\circ\bullet]; p, [\circ\bullet\circ]; j, [\circ \bullet \circ]))$

$= ((j; q)(k; e; w)(q)(k; p; j))$

merge$(lc, l, p)$$\rightarrow$ **list of couples** returns the empty list if lc and l are both empty and

$$\bigcup_{\substack{\langle i,c\rangle \in lc \\ i \in l}} (\langle i, c \vee colors(p)\rangle) \cup \bigcup_{\substack{\langle i,*\rangle \notin lc \\ i \in l}} (\langle i, colors(p)\rangle)$$

otherwise.

Example: assuming $\delta_i^- = 3$,

$merge((j, [\circ \bullet \circ]; q, [\circ \circ \bullet]; k, [\circ \bullet \circ]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]), (q; k; p; j), 1)$

$= (j, [\bullet \bullet \circ]; q, [\bullet \circ \bullet]; k, [\bullet \bullet \circ];\ e, [\circ \circ \bullet];\ w, [\circ \circ \bullet]; p, [\bullet \circ \circ])$

new$(lc, l)$$\rightarrow$ **list of couples** returns the empty list if lc is empty and the list of pairs $\langle identifier,\ colors\rangle$ whose $identifier$ is not in l otherwise.

Example: assuming $\delta_i^- = 3$,

$new((j, [\circ \bullet \circ]; q, [\circ \circ \bullet]; k, [\circ \bullet \circ]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]), (i; j; e))$

$= (q, [\circ \circ \bullet]; k, [\circ \bullet \circ];\ w, [\circ \circ \bullet])$

problem$(Llc)$$\rightarrow$**boolean** returns *true* if there exist two integers p and q such that $p \leq \sharp(Llc)$ and $q \leq \sharp(Llc[p])$ and $Llc[p][q]$ is of the form $\langle identifier, colors\rangle$ and all booleans in *colors* are *false* ($\circ$). Otherwise, this function returns *false.*

3.4. The Algorithm

In addition to its local memory, each node makes use of the following local variables when processing messages: α is the current index in the local memory and message main list, *i_pertinent* is a boolean that is *true* if the α^{th} element of the local memory main list contains pertinent information, *m_pertinent* is a boolean that is *true* if the α^{th} element of the message main list contains pertinent information, *known* is the list of all identifiers found in the local memory and message found up to rank α, *temp* is a temporary list that stores the updated α^{th} element of the local memory main list.

We are now ready to present our Census Algorithm (noted $\mathcal{CA}$ in the remaining of the paper). This algorithm is message driven: processors execute their code when they receive an incoming message. In order to perform correctly in configurations where no messages are present, Algorithm $\mathcal{CA}$ also uses a spontaneous action that will emit a message.

Spontaneously, a node i runs the following code:

```
IF $problem(local\_memory)$ THEN $local\_memory := ()$ ENDIF
$emit(i, local\_memory)$
```

Upon receipt of a message named *message* from incoming link number p, a node i whose local memory is denoted by *local_ memory* runs the following code:

```
**Check for local memory coherence:**
$i\_pertinent :=$ NOT $problem(local\_memory)$
**Check for message coherence:**
$m\_pertinent := (\sharp(message[1]) = 1)$
**Update local memory:**
$\alpha := 0; known := (i);$
WHILE $m\_pertinent$ OR $i\_pertinent$ DO
    $\alpha := \alpha + 1; temp := ()$
    $local\_memory[\alpha] := clean(local\_memory[\alpha], p)$
    IF $i\_pertinent$ THEN
        $temp := new(local\_memory[\alpha], known)$
        IF $temp = ()$ THEN $i\_pertinent :=$ FALSE ENDIF
    ENDIF
    IF $m\_pertinent$ THEN
        IF $message[\alpha] \setminus known = ()$ THEN
            $m\_pertinent :=$ FALSE
        ELSE
            $temp := merge(temp, message[\alpha] \setminus known, p)$
        ENDIF
    ENDIF
    IF $temp \neq ()$ THEN
        $local\_memory[\alpha] := temp$
        $known := known \cup identifiers(temp)$
    ENDIF
ENDWHILE
**Truncate local memory up to position $\alpha$ :**
$local\_memory := (local\_memory[1], \ldots, local\_memory[\alpha])$
**Emit message along with identifier:**
$emit(i, local\_memory)$
```

4. PROOF OF CORRECTNESS

In this section, we show that Algorithm $\mathcal{CA}$ is a self-stabilizing Census algorithm. In more details, independently of the initial configuration of network channels (non infinitely full) and of the initial configuration of local memories of nodes, every node ends up with a local memory that reflect the contents of the network, even if unreliable communication media is used for the underlying communication between nodes.

4.1. Overview

First, we define a formal measure on messages that circulate in the network and on local memories of the nodes. This measure is either the distance between the current form of the message and its canonical form (that denotes optimal knowledge about the network), or between the current value of the local memories and their canonical form (when a node has a perfect knowledge about the network). We use this measure to compute the weight of a configuration.

Then, we show that after a set of emissions and receptions of messages, the weight of a configuration decreases. An induction shows that this phenomenon continue to appear and that the weight of a configuration reaches 0, *i.e.* a configuration where each message is correct and where each node has an optimal knowledge about the network. We also show that such a configuration (whose weight is 0) is stable when a message is emitted or received. According to the previous definitions, a configuration of weight 0 is a *legitimate configuration* after finite time.

These two parts prove respectively the convergence and closure of our algorithm, and establish its self-stabilizing property.

4.2. Legitimate Configurations

The Census problem being static and deterministic, when we only consider node local memories, there is a single legitimate configuration. This legitimate configuration is when each node has a global correct knowledge about the network. It is also the stable configuration the system would reach had it been started from a zero knowledge configuration (where the local memory of each node is null, and where no messages are in transit in the system).

In this legitimate configuration, all circulating messages are of the same kind, and the only difference between legitimate configurations is the number of messages in each communication link. This induces the following definitions of canonical messages and canonical local memory.

Definition 5. The canonical form of a message circulating on a link between nodes j and i is the list of lists starting with the singleton list (j) followed by the χ_j lists of ancestors of j at distance between 1 and χ_j.

Definition 6. The canonical form of node i's local memory is the list of lists of pairs *Llc* of the χ_i lists of pairs $\langle$*identifier*,*colors*$\rangle$ such that:

- *identifiers*(*Llc*) is the list of the χ_i lists of ancestors of i at distance 1 to χ_i.
- if a shortest path from node j to node i passes through the p^{th} input channel of i, then the boolean array *colors* associated to node j in *Llc* has *colors*$[p] = \bullet$.

For the sake of simplicity, we will also call the α^{th} list of a canonical message or a canonical local memory a *canonical list.*

4.3. Closure

Proposition 1. *The canonical form of node i's local memory and that of its incoming and outgoing channels are coherent.*

Proof. If node i's local memory is in canonical form (according to Definition 6), then the *emit* action trivially produces a canonical message (according to Definition 5).

Conversely, upon receipt by node i of a canonical message through incoming link j, the local memory of i is replaced by a new identical canonical memory. Indeed, *clean* first removes from the α^{th} list of i's local memory all pairs $\langle$*identifier*, *colors*$\rangle$ such that *colors*= *colors*(p), yet by definition of canonical memory, each such *identifier* is that of a node such that the shortest path from *identifier* to i is of length α and passes through j. Moreover, the list l used by *merge* is the list of nodes at distance $\alpha - 1$ of node i, so for any *identifier* appearing in l, two cases may occur:

1. There exists a path from *identifier* to i that is of length $< \alpha$, then *identifiers*$\in$ *known* and it does not appear in the new list of rank α,
2. There exists a shorter path from *identifier* to i through j of length α, then $\langle$*identifier*, *colors*(p)$\rangle$ is one of the elements that were removed by *clean* and this information is put back into node i's local memory.

∎

Corollary 1. *The set of legitimate configurations is closed.*

Proof. Starting from a configuration where every message and every local memory is canonical, none of the local memories is modified, and none of the emitted message is non-canonical. ∎

4.4. Configuration Weight

We define a weight on configurations as a function on system configurations that returns a positive integer. As configurations of weight zero are legitimate, the weight of a configuration c denotes the "distance" from c towards a legitimate configuration.

In order to evaluate the weight of configurations, we define a measure on messages and local memory of nodes as an integer written using $D + 2$ digits in base 3 (where D denotes the graph diameter). The weight of a configuration is then the pair of the maximum weight of local memories, and the maximum weight of circulating messages. For sake of clarity, a single integer will denote the weight of the configuration when both values are equal. Note that since a canonical message is of size $\leq D + 1$, we have $m\_canonical[D + 2] = ()$.

Definition 7. Let *mess* be a circulating message on a communication link whose canonical message is denoted by *m_ canonical*. The weight of *mess* is the integer written using $D + 2$ base 3 digits and whose α^{th} digit is:

- 0, if $mess[\alpha] = m\_canonical[\alpha]$,
- 1, if $mess[\alpha] \subsetneq m\_canonical[\alpha]$,
- 2, if $mess[\alpha] \not\subseteq m\_canonical[\alpha]$.

Example : Assume that a link from j to i in a network of diameter 5 has a canonical message of the form $(j)(k;e;w)(q)(i;p)$. The following messages circulating on this channel will have the following weights:

(j)	$(k;e;w)$	(q)	$(i;p)$				
0	0	0	0	0	0	0	Overall a weight of 0
(j)	$(k;e;w)$	$(q;d)$	$(i;p)$	(z)			
0	0	2	0	2	0	0	Overall a weight $\leq 3^5$
(j)							
0	1	1	1	1	0	0	Overall a weight $\leq 2 \times 3^5$
(g)	$(h;t)$	(t)	$(i;d)$	(a)	(a)	(a)	
2	2	2	2	2	2	2	Overall a weight of $3^7 - 1$

Then, $3^{D+2}-1$ is the biggest weight for a message, and corresponds to a message that is totally erroneous. At the opposite, 0 is the smallest weight for a message, and corresponds to a canonical message, or to a message that begins with a canonical message.

Definition 8. Let *memo* be the local memory of a node i whose canonical local memory is *m_ canonical.* The weight of *memo* is the integer written using $D+1$ digits (in base 3) and whose α^{th} digit is:

- 0, if $memo[\alpha] = m\_\ canonical[\alpha]$
- 1, if $memo[\alpha] \neq m\_\ canonical[\alpha]$ and $identifiers(memo[\alpha]) \subseteq identifiers($ $m\_\ canonical[\alpha])$ and for any $\langle$ *identifier*, $colors_1$ $\rangle$ of $memo[\alpha]$, the associated $\langle$ *identifier*,$colors_2\rangle$ in $m\_\ canonical[\alpha]$ verifies: $(colors_1 \wedge colors_2) = colors_1$.
- 2, otherwise.

Then $3^{D+1} - 1$ is the biggest weight of a local memory and denotes a totally erroneous local memory. At the opposite, 0 is the smallest weight and denotes a canonical local memory.

Let us notice that in both cases (weight of circulating messages and of nodes local memories), the α^{th} digit 0 associated to the α^{th} list denotes that this particular list is in its final form (the canonical form). The α^{th} digit 1 means that the α^{th} list is coherent with the α^{th} canonical list, but still lacks some information. On the contrary, the α^{th} digit 2 signals that the related α^{th} position contains informations that shall not persist and that are thus unreliable. The weight of a message indicates how much of the information it contains is pertinent.

4.5. Convergence

After defining message weight and, by extension, configuration weights, we first prove that starting from an arbitrary initial configuration, only messages of weight lower or equal to $3^{D+1} - 1$ are emitted, which stands for the base case for our induction proof.

Lemma 1. *In any configuration, only messages of weight lower than* 3^{D+1} *may be emitted.*

Proof. Any message that is emitted from a node i on a link from i to j is by function *emit*. This function ensures that this message starts with the singleton list (i). This singleton list is also the first element of the canonical message for this channel. Consequently, the biggest number that may be associated to a message emitted by node i starts with a 0 and is followed by $D+1$ digits equal to 2. Its overall weight is at most $3^{D+1}-1$. ∎

LEMMA 2. *Assume $\alpha \geq 1$. The set of configurations whose weight is strictly lower than $3^{\alpha-1}$ is an attractor for the set of configuration whose weight is strictly lower than 3^{α}.*

Proof. A local memory of weight strictly lower that 3^{α} contains at most α erroneous lists, and it is granted that it starts with $D+2-\alpha$ canonical lists.

By definition of the *emit* function, each node i that owns a local memory of weight strictly below 3^{α} shall emit the singleton list (i) followed by $D+2-\alpha$ canonical lists. Since canonical messages sent by a node and its canonical local memory are coherent, it must emit messages that contain at least $D+2-\alpha+1$ canonical lists, which means at worst $\alpha-1$ erroneous lists. The weight of any message emitted in such a configuration is then strictly lower than $3^{\alpha-1}$.

It follows that messages of weight exactly 3^{α} which remain are those from the initially considered configuration. Hence they are in finite number. Such messages are either lost or received by some node in a finite time. The first configuration that immediately follows the receiving or loss of those initial messages is of weight (3^{α} (local memory), $3^{\alpha-1}$ (messages)).

The receiving by each node of at least one message from any incoming channel occurs in finite time. By the time each node receives a message, and according to the local memory maintenance algorithm, each node would have been updated. Indeed, the receiving of a message from an input channel implies the cleaning of all previous information obtained from this channel. Consequently, in the considered configuration, all lists in the local memory result from corresponding lists in the latest messages sent through each channel. Yet, all these latest messages have a weight strictly lower than $3^{\alpha-1}$ and by the coherence property on canonical forms, they present information that are compatible with the node canonical local memory, up to index $D+3-\alpha$. By the same property, and since all input channels contribute to this information, it is complete. In the new configuration, each node i maintains a local memory whose first $D+3-\alpha$ lists are canonical, and thus the weight of its local memory is $3^{\alpha-1}$. Such a configuration is reached within finite time and its weight is ($3^{\alpha-1}$ (local memory), $3^{\alpha-1}$ (messages)). ∎

PROPOSITION 2. *The set of configurations whose weight is* 0 *is an attractor for the set of all possible configurations.*

Proof. By induction on the maximum degree of the weight on configurations. The base case is proved by Lemma 1, and the induction step is proved by Lemma 2. Starting from any initial configuration whose weight is greater that 1, a configuration whose weight is strictly inferior is eventually reached. Since the weight of a configuration is positive or zero, and that the order defined on configurations weights is total, eventually a configuration whose weight is zero is eventually reached. By definition, this configuration is legitimate. ∎

THEOREM 1. *Algorithm $\mathcal{CA}$ is self-stabilizing.*

Proof. Consider a message m of weight 0. Two cases may occur: *(i)* m is canonical, or *(ii)* m starts with a canonical message, followed by at least one empty list, (possibly) followed by several erroneous lists. Assume that m is *not* canonical, then it is impossible that m was emitted, since the **truncate** part of Algorithm $\mathcal{CA}$ ensures that no message having an empty list can be emitted; then m is an erroneous message that was present in the initial configuration.

Similarly, the only local memories that may contain an empty list are those initially present (*e.g.* due to a transient failure).

As a consequence, after receipt of a message by each node and after receipt of all initial messages, all configurations of weight 0 are legitimate (they only contain canonical messages and canonical local memories).

By Proposition 2, the set of legitimate configurations is an attractor for the set of all possible configurations, and Corollary 1 proves closure of the set of legitimate configurations. Therefore, Algorithm $\mathcal{CA}$ is self-stabilizing. ∎

4.6. Complexity

In this section, we investigate the memory space and time needed for the system to stabilize into a legitimate configuration.

4.6.1. Space complexity

The space complexity result is immediately given by the assumptions made when writing our algorithm. In the following, N denotes the number of nodes in the system, and k denotes the number of possible identifiers for nodes. In practical systems, $\log_2(k)$ typically corresponds to a system word (32 or 64 bits).

Lemma 3. *Each message m requires at least $N \times (\log_2(k))$ bits space.*

Proof. We compute the space needed by each message to hold all information in the Census algorithm. We do not take into account the implementation dependent list coding of a message information. Each identifier (of size bounded by $\log_2(k)$) is present exactly once in each message, and there are N such identifiers. Overall, the required memory (in bits) at message m is bounded by:

$$N \times (\log_2(k))$$

∎

Lemma 4. *Each node i requires at least*

$$(N-1) \times \left(\log_2(k) + \delta_i^-\right)$$

bits space, where δ_i^- denotes the input degree of the node i.

Proof. We compute the space needed at node i to hold all information in the Census algorithm. We do not take into account the implementation dependent list coding of a node local information. Each identifier (that is bounded by $\log_2(k)$) in a pair is associated to an boolean array, that represents the incoming links that transmitted the identifier. This array requires δ_i^- bits. In a correct configuration, node i has a pair $\langle$*identifier*, *colors*$\rangle$ for any other node in the network (thus $N-1$ pairs). Overall, the required memory (in bits) at node is bounded by:

$$(N-1) \times \left(\log_2(k) + \delta_i^-\right)$$

where δ_i^- denotes the input degree of the node i. ∎

4.6.2. Time complexity

In the convergence part of the proof, we only assumed that computations were maximal, and that message loss, duplication and desequencing could occur. In order to provide an upper bound on the stabilization time for our algorithm, we assume strong synchrony between nodes and a reliable communication medium between nodes. Note that these assumptions are used for complexity results only, since our algorithm was proven correct even in the case of asynchronous unfair computations with link intermittent failures. In the following D denotes the network diameter.

LEMMA 5. *Assuming a synchronous reliable system $\mathcal{S}$, the stabilization time of algorithm $\mathcal{CA}$ is $O(D)$.*

Proof. Since the network is synchronous, we consider *system steps* as: *(i)* each node receives all messages that are located at each of its incoming links and updates its local memory according to the received information, and *(ii)* each node sends as many messages as received on each of its outgoing links. Intuitively, within one system step, each message is received by one processor and sent back. Within one system step, all messages are received, and messages of weight strictly inferior to that of the previous step are emitted (see the proof of Lemma 2). In the same time, when a processor has received messages from each of its incoming links, its weights is bounded by $3^{D+1-\alpha}$, where D is the network diameter, and α is the number of the system step (see the proof of Lemma 2). Since the maximal initial weight of a message and of a local memory is 3^{D+2}, after $O(D)$ system steps, the weight of each message and of each local memory is 0, and the system has stabilized. ∎

5. CONCLUSION

When a distributed system is subject to various kinds of failures, various ways of ensuring recovery from those failures are to be considered. We presented a global approach that allows to solve the Census problem while tolerating two usually distinguished kinds of failures: transient memory failures, and intermittent link failures. Unlike previous work, we did not specifically address the problems related to intermittent link failures. More precisely, in the proof of correctness, we presented a global weight function, and showed that in any system computation, its value was strictly decreasing up to the point when it reached 0. In this final stage, a key property related to idempotency (the receipt of a correct message by a correct node does not modify its state, a correct node always sends the same correct message) hints at a possible general condition for tolerating both transient and intermittent link failures.

We considered the Census problem in a strongly connected graph. However, the very same algorithm can be used for different purposes. In [18], Dolev and Herman show that starting from an algorithm that simply collects node unique identifiers, communicating other local information as well as the node identifier leads to solutions for any silent task (see [17]) using the same underlying Census algorithm. For example, storing local topology information enables the construction of a Topology Update algorithm. In our context, mixing the approach of [18] and ours would lead to a self-stabilizing Topology Update algorithm that also supports link intermittent failures.

Although we assume the system communication graph to be strongly connected, we do not use this information to build a well known topology (*e.g.* a ring) and run a well known algorithm on it. Indeed, this approach could potentially lower down the performance of the overall algorithm, due to the fact that the communication possibilities are not used to their full extent. As a matter of fact, when our distributed algorithm is run on a network that is not strongly connected, we ensure that the collected information at each node is a topologically sorted list of its ancestors. In DAG (directed acyclic graph) structured networks, such kind of information is often wished (see [11]), and our approach makes it tolerant to link failures for free.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for helping us to improve this paper, both in presentation and in technical correctness.

REFERENCES

[1] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 4(3):1–48, 1998.

[2] Y Afek and G M Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[3] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990.

[4] E Anagnostou and V Hadzilacos. Tolerating transient and permanent failures. In *Proceedings of WDAG'93, LNCS 725*, pages 174–188, 1993.

[5] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In IEEE, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 268–277, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.

[6] K A Bartlett, R A Scantlebury, and P T Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[7] A Basu, B Charron-Bost, and S Toueg. Simulating reliable links in the presence of process crashes. In *Proceedings of the Tenth International Workshop on Distributed Algorithms (WDAG'96), LNCS 1151*, 1996.

[8] J Beauquier, AK Datta, and S Tixeuil. Self-stabilizing census with cut-through constraint. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 70–77. IEEE Computer Society, 1999.

[9] J Beauquier and Kekkonen-Moneta. Fault tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems science*, 28(11):1177–1187, november 1997.

[10] A Bui, A K Datta, F Petit, and V Villain. State-optimal snap-stabilizing pif in tree networks. In *Proceedings of the Fourth Workshop on Self-stabilizing Systems*, pages 78–85, 1999.

[11] S K Das, A K Datta, and S Tixeuil. Self-stabilizing algorithms in dag structured networks. *Parallel Processing Letters*, 9(4):563–574, December 1999.

[12] S Delaët and S Tixeuil. Un algorithme auto-stabilisant en dépit de communications non fiables. *Technique et Science Informatiques*, 5(17), 1998.

[13] S Delaët and S Tixeuil. Tolerating transient and intermittent failures. In *Proceedings of OPODIS'2000*, pages 17–36, December 2000.

[14] E W Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[15] S Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42(2):122–127, May 1997.

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

[17] S Dolev, MG Gouda, and M Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[18] S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

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

[20] M J Fisher, N A Lynch, and M S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[21] A. S. Gopal and K. J. Perry. Unifying self-stabilization and fault-tolerance. In *Proceedings of PODC'93*, pages 195–206, 1993.

[22] J-H. Hoepman, M. Papatriantafilou, and P. Tsigas. Self-stabilization of wait free shared memory objects. In *Proceedings of the WDAG'95*, pages 273–287, 1995.

[23] M Jayaram and G Varghese. Crash failures can drive protocols to arbitrary states. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 247–256, 1996.

[24] N A Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[25] T Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.

[26] J M Spinelli and R G Gallager. Event driven topology broadcast without sequence numbers. *IEEE Transaction on Communications*, 37(5):468–474, May 1989.

[27] N V Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, September 1976.

[28] D-W Wang and L D Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, pages 73–83, August 1989.

SYLVIE DELAËT is *Docteur en Sciences de l'Université Paris Sud*, Orsay, France since December 1995. Her Ph.D. was on self-stabilizing mutual exclusion. She has been a *Maître de conférences* at the University Paris Sud since September 1996. From 1995 to 2001 she was doing research in the self-stabilizing area. Her research interests include distributed computing, communication networks and fault tolerance.

SÉBASTIEN TIXEUIL received the *Magistère d'Informatique Appliquée* from the University Pierre and Marie Curie (France) in 1995, and his M.Sc and Ph.D. in Computer Science from the University of Paris Sud (France) in 1995 and 2000, respectively. In 2000, he joined the faculty at the University Paris Sud. His research interests include self-stabilizing and fault-tolerant distributed computing.