Fair and Reliable Self-Stabilizing Communication

Colette Johnen, Ivan Lavallee, Christian Lavault

Fair and Reliable Self-Stabilizing Communication

Colette JOHNEN [a] Ivan LAVALLÉE [b] Christian LAVAULT [c]*

[a] *LRI-CNRS Université Paris-Sud*

[b] *LRIA-Paradis, Université Paris 8*

[c] *LIPN, CNRS UPRES-A 7030, Université Paris-Nord*

Abstract

We assume a link-register communication model under read/write atomicity, where every process can read from but cannot write into its neighbours' registers. The paper presents two self-stabilizing protocols for basic fair and reliable link communication primitives. The first primitive guarantees that any process writes a new value in its register(s) only after all its neighbours have read the previous value, whatever the initial scheduling of processes' actions. The second primitive implements a "weak rendezvous" communication mechanism by using an alternating bit protocol: whenever a process consecutively writes n values (possibly the same ones) in a register, each neighbour is guaranteed to read each value from the register at least once.

Both protocols are self-stabilizing and run in asynchronous arbitrary networks. The goal of the paper is in handling each primitive by a separate procedure, which can be used as a "black box" in more involved self-stabilizing protocols.

Keywords: Self-stabilization, communication primitive, read/write atomicity, rendezvous, liveness

1 Introduction

A *self-stabilizing* system which is started from an arbitrary initial configuration, regains its consistency and demonstrates legal behaviour by itself, without any outside intervention. Consequently, a self-stabilizing system needs not be initiated to any configuration, and can recover from *transient faults*. More precisely, it can recover from *memory corruptions* and copes with processors or channels crashes and recoverings (i.e., dynamic networks).

1.1 The Communication primitives

In the paper (see also [19]), we present fair and reliable self-stabilizing communication primitives in the link-register model. Communication between two neighbours (A and B) is carried out by the use of two sets of *communication registers* called *registers*: r_{AB} and r_{BA}. Process A can write in the registers of r_{AB} and each process A and B can read from the registers of r_{AB}. The registers support *read* and *write* atomic operations.

The communication primitives allow any process which writes a value in its own register (say $Write$) to make sure that every neighbour eventually reads that latter value before writing another value in the $Write$ register, whatever the initial scheduling of processes' actions. The self-stabilizing protocols for these basic communication primitives run on dynamic asynchronous arbitrary networks.

The first primitive guarantees that any process A writes a new value in its register(s) $Write_{AB}$ only after its neighbour B has read the previous value. Notice that when A writes n times the same value

*Corresponding author: LIPN, CNRS UPRES-A 7030, Université Paris-Nord, 99, Av. J.-B. Clément 93430 Villetaneuse, France. Email: lavault@lipn.univ-paris13.fr consecutively in the register $Write_{AB}$, the primitive ensures that B eventually copies this value at least once. This primitive simulates self-stabilizing reliable message-passing communications in the link-register asynchronous model. It guarantees that a message, that is the value of the register $Write$, is eventually received: the value is eventually known from the neighbours' process.

The *rendezvous* mechanism (as defined in [16]) synchronizes communications, i.e., the *write* and *read* operations are performed in and from the same register. When Process A writes a value in its register $Write_{AB}$, it cannot perform any other action until process B has completed a *read* operation from the register $Write_{AB}$. The second communications primitive is a self-stabilizing "weak rendezvous". After performing a *write* operation in its register $Write_{AB}$, the process A cannot perform but some specific actions, as long as process B has not completed a *read* operation from $Write_{AB}$. Therefore, if A consecutively writes n values (possibly the same ones) in the register $Write_{AB}$, the primitive guarantees that B eventually copies each value at least once. Incidentally, this protocol also maintains a weak scheduling between processes in arbitrary networks: if A writes n times the same value in $Write_{AB}$, the value will be read at least n times.

Each such very basic primitive may prove useful as a communication "black box" in designing more involved distributed self-stabilizing protocols.

1.2 Related Works en Results

A deterministic self-stabilizing "balance-unbalance" mechanism on two processes systems under read/write atomicity is presented in [12] and in [13]. The two processes are not executing the same code. The one executes the balance code: when both processes have the same color, it changes color. The other executes the unbalance code: when both processes have not the same color, it changes color. In [12], this mechanism is used to guarantee that each process has a mutual exclusion access to a critical section, and in [13], it is used to ensure synchronization of the processes. In both cases, this mechanism provides strong synchronization: between two "actions" of a process, the other process cannot perform but only one "action". In [12, 13], the two processes protocol is used to design a mutual exclusion algorithm (global synchronization) on tree networks. As claimed in [12, 13], the balance-unbalance mechanism cannot be extended to any network topology, since there exist no deterministic self-stabilizing synchronization protocols in uniform arbitrary networks. On the other hand, a self-stabilizing synchronization on unidirectional rings is provided in [10] through the deterministic token circulation mechanism: between two actions of a process its neighbours cannot perform but only one action.

Any self-stabilizing reset protocol [5, 2, 8] can be combined with the protocol in [6] to design a self-stabilizing synchronizer. General self-stabilizing synchronizers are presented e.g. in [9, 7, 20]. Global self-stabilizing synchronizers for tree networks are also proposed in [13, 3, 11]. A self-stabilizing local synchronizer, that synchronizes each node in a tree network with its neighbours is presented in [18].

In [4], Anagnostou and Hadzilacos present a self-stabilizing data link protocol under the read/write atomicity model. Their protocol uses the balance-unbalance mechanism in order to synchronize operations performed on a register (between two *write* operations in the register there is only one *read* operation from that register). No proof of the protocol is given in [4]. For instance, the authors do not explain how the data link protocol can guarantee starvation-freeness.

By contrast, our primitives use no balance-unbalance mechanism (but the alternating-bit mechanism in the second primitive). Thus, they can be used in any network topology.

In the recent literature, several communication problems in the message-passing model have been addressed. A self-stabilizing communication protocol for two-way handshake is presented in [15], and a self-stabilizing version of the alternating-bit protocol is given in [1].

Section 2 describes our model with the basic assumptions. In Section 3, we present the general principle of our solution for a two processes system. The generalization to n processes in arbitrary networks yields the *Read Checking* self-stabilizing protocol, which is presented in Section 4. Section 5 is devoted to the proof of liveness and correctness of the Read Checking protocol. Section 6 presents the weak rendezvous protocol. Finally, the paper ends with few concluding remarks.

2 Model and Requirements

Although distinct from the one described in [12], our model relies on close requirements and assumptions, especially in terms of communication (e.g., link registers, read/write atomicity, etc.). A distributed system consists of n processes denoted A, B, etc. Each process resides on a node of the system's *communication graph* (or *network*). Two processes which reside on two adjacent nodes of the network are called *neighbours*. We model distributed self-stabilizing systems as a set of (possibly infinite) state machines called processes. Each process can only communicate with the subset of processes consisting of its neighbours. We assume a *link-register* communication model under read/write atomicity [12]. Each link between any two neighbours A and B is composed of two pairs of registers[1], denoted $(Write_{AB}, Read_{AB})$ and $(Write_{BA}, Read_{BA})$, and belonging to A and B, respectively. Process A can read from the two registers of B, $Write_{BA}$ and $Read_{BA}$, but cannot write into them. Similarly, process A cannot write but in its own registers, $Write_{AB}$ and $Read_{AB}$, to communicate with B.

A *configuration* of the system is the vector of states of all processes. The state of a process is the value of its internal variables and the contents of its registers.

2.1 Schedulers, Demons and Computation

An *atomic step* is the "largest" step which is guaranteed to be executed uninterruptedly. A process uses *read/write* atomicity if each atomic step contains either a single read operation or a single write operation but not both. The system behaviour is modelled by the interleaving model in which processes are activated by a scheduler. The scheduler is regarded as a *fair* adversary: in a self-stabilizing system, all possible fair executions are required to converge to a correct behaviour. A fair scheduler shall eventually activate any process which may continuously perform an action. A common scheduler activates either processes one by one (central demon) or subsets of processes (distributed demon). Under read/write atomicity, both central and distributed schedulers/demons are "equivalent", in the sense that any execution performed under a distributed scheduler may be simulated by a central one. A process which can perform an atomic step into a configuration c, is said to be *enabled* at c. During a *computation step*, one or more processes execute an atomic step. A *computation* of a protocol $\mathcal{P}$ is a sequence of configurations $c_1, c_2, \ldots$ such that, for $i = 1, 2, \ldots$, the configuration c_{i+1} is reached from c_i by one computation step. A computation is said to be *maximal* either if the sequence is infinite, or if it is finite and no process is enabled in the final configuration. A *problem* is a predicate defined on computations.

2.2 Self-Stabilization

The protocol $\mathcal{P}$ is *self-stabilizing* for the problem Π if and only if there exists a predicate $\mathcal{L}$ defined on configurations such that:

- all computations reach a configuration that satisfies $\mathcal{L}$ (**convergence**);
- all computations, from $\mathcal{L}$, satisfy problem Π (**correctness**).

Notice that the maximal computations of a self-stabilizing protocol may be finite; in that case the algorithm is said to be *silent* [14]. Most self-stabilizing algorithms which build spanning tree or elect a leader are silent [17]. Self-stabilizing protocols offers full and automatic protection against all transient

[1]In our model, the registers are physical (hardware) devices. Reading from or writing in *one* register is an atomic action according to the design of the microprocessor.

process failures, no matter how much the data have been corrupted: e.g., all registers values may be fully corrupted.

So, whatever the registers values, our protocols secure the transfer of information between any two pair of neighbours after a "certain delay time".

3 Principle of the Solution

Let a two processes system, consisting in two neighbouring processes A and B equipped with their two pairs of registers (see Section 2). The principle of the solution for A relies on the following basic idea. Under read/write atomicity, A systematically keeps reading the value from $Write_{BA}$ and copies out this value in $Read_{AB}$ (i.e., A reads the message sent by B and copies out the message in $Read_{AB}$ to inform B that its message is received). Besides, A systematically keeps reading the value from $Read_{BA}$ and compares it to the value of $Write_{AB}$. When both values are equal, A finds out that B somehow read that value (i.e., the information has been transmitted), So it can stop reading and can write again in $Write_{AB}$.

```
while true do
        $A$ writes in $Write_{AB}$
    repeat
        $A$ reads from $Write_{BA}$ ;
        $A$ writes out the value of $Write_{BA}$ into $Read_{AB}$ ;
        $A$ reads from $Read_{BA}$
    until $Read_{BA} = Write_{AB}$
endwhile
```

Fig. 1. *The basic 2-processes protocol for A.*

After A has written a new value in $Write_{AB}$, A becomes "weakly locked" until B receives the message ($Read_{BA} = Write_{AB}$). When A is inside the **repeat** loop, it can only perform some actions, for instance, A cannot write in its register $Write_{AB}$.

In a self-stabilizing setting, A may then proceed with the execution of its own code, since the protocol makes it sure that B did read the value from $Write_{AB}$ (at least, it results from the protocol that A knows for sure that the values in $Read_{BA}$ and $Write_{AB}$ are identical). The corresponding code sequence for B is of course fully symmetrical to the basic protocol for A: the roles of A and B (i.e. the registers' names) have simply to be inverted within the above protocol in Fig. 1. Thus, a two-way communication is established between A and B.

4 The Protocol in Arbitrary Networks

The generalization of the above protocol to a system of $n > 2$ processes constituting an arbitrary network is now easy. We still assume each pair of neighbouring processes in the network to be equipped with its two pairs of registers on their common link. In order to simplify the use of variables, we call *"message"* the "information" exchanged between neighbours during the execution of the protocol.

A protocol which stabilizes on a single link may not generalize to a protocol which stabilizes on all links of a (finite) network, e.g. by having each process execute the "link-protocol" in a round robin manner on each individual link adjacent to it. Taking the n-processes system pair by pair may cause a deadlock: for all $i \in \{0, \ldots, n-1\}$, A_i may be waiting for A_{i+1} to read from $Write_{A_iA_{i+1}}$, with $A_n = A_0$.

4.1 Notation

Write register for A: $Read_{AB_i}$ is the register in which A writes the value of the last message read by A and sent by B_i.

Read register for A: $Write_{B_iA}$ is the register in which B_i writes the message to be transmitted to A, and $Read_{B_iA}$ is the register in which B_i writes the value of the last message read by B_i and sent by A.

Write and read register for A: $Write_{AB_i}$ is the register in which A writes the value of the message which is to be sent to its ith neighbour B_i.

Function get_i for A: get_i takes no argument and returns the next message to be sent to the ith neighbour of A (get_i is a helper function added to A).

4.2 The Read Checking Protocol

On the same assumptions for the model (read/write atomicity) and for the scheduler's actions (rules of activations of processes and fairness) as given in Section 2, the specification of the self-stabilizing Read Checking protocol in arbitrary networks for a process A, with neighbours B_i's ($1 \le i \le N_A$), is as follows.

```
constant $N_A$      : the number of neighbours of $A$ ;
var  $s_i$          : message to be sent to the $i$th neighbour of $A$ ;
     $r_i$          : message sent from the $i$th neighbour of $A$ ;
     $val_i$        : value of the last message sent from $A$ and read by the $i$th neighbour of $A$ ;

while $true$ do
   for $i = 1$ to $N_A$ do
         write($Write_{AB_i}, get_i$) ;
   endfor
   repeat
      for $i = 1$ to $N_A$ do
         $r_i \leftarrow$ read($Write_{B_iA}$) ;
         write($Read_{AB_i}, r_i$) ;
         $val_i \leftarrow$ read($Read_{B_iA}$) ;
         $s_i \leftarrow$ read($Write_{AB_i}$) ;
      endfor
   until ( $\forall i \in [1, N_A]$  $val_i = s_i$ )
endwhile
```

Fig. 2. *The Read Checking protocol for A.*

5 Proof of the Read Checking Protocol

5.1 Proof of Liveness

Lemma 5.1 *Let γ be any configuration of an arbitrary network of processes on which the read checking protocol is performed. All processes are enabled in configuration γ.*

Proof. Let A be a process, its program counter is such that

• A is not in the **repeat** loop, and hence A can write into one of its *Write* registers;

• A is in the **repeat** loop, and hence A can either read from one of its neighbours' register, or write into one of its *Read* registers. Thus, in all configuration, A can perform an atomic step (if chosen by the scheduler). □

Lemma 5.2 *Every execution of the protocol on any arbitrary network is infinite.*

Proof. From Lemma 5.1, whatever the current configuration, all processes can execute an action. Hence, every configuration is deadlock-free and no execution can reach a deadlock configuration. Therefore, every execution is infinite. □

Lemma 5.3 *Whatever the execution, every process performs an infinite number of actions.*

Proof. From Lemma 5.2, every execution is infinite. From Lemma 5.1, in each configuration that is reached every process can perform an action. The scheduling of processes' actions is fair: if a process can always execute an action, then the process finally performs an action. Thus, by fairness, every process is performing an infinite number of actions, whatever the execution. □

Definition 5.1 *Let A and B be two neighbouring processes. A is said to allow B to write iff $Read_{BA} = Write_{AB}$. Let A be a process and let N_A denote the number of neighbours of A (N_A is the degree of A in the network).*

Definition 5.2 *Let A and B be two neighbouring processes. The update of the register $Read_{AB}$ is the sequence of the two following actions performed by B: $r_i \leftarrow$ **read**$(Write_{AB})$; **write**$(Read_{BA}, r_i)$.*

A wrong *writing is a write action in the register $Read_{BA}$ which is not performed within the context of an update.*

The correct *writing into the register $Read_{BA}$ is a write action executed within the context of an update.*

Lemma 5.4 *Let A be a process with its program counter in the* **repeat** *loop and let B be a neighbour of A. Whatever the current configuration and the execution, the processes system executing the protocol either eventually reaches a configuration in which B allows A to write, or A exits the* **repeat** *loop.*

Proof. Suppose B never allows A to write and A never exits the **repeat** loop. Then A never changes the value in its register $Write_{AB}$. Under these conditions, updating its register $Read_{BA}$ is a writing permission given to A by B (since between the reading of the value from the register $Write_{AB}$ and the writing of that value in $Read_{BA}$, the register $Write_{AB}$ does not change value).

Whatever the current configuration and the execution, if the program counter of B is not within the **repeat** loop, it takes B less than N_B actions to enter the **repeat** loop. Once B enters the loop, after $4N_B$ actions, it updates all its *Read* registers, and thus allows A to write.

Whatever the current configuration and the execution, if the program counter of B is within the **repeat** loop, it takes B at least $4N_B$ actions either to exit the loop, or to update its register $Read_{AB}$.

Whatever the execution, B performs an infinite number of actions (by Lemma 5.4) and eventually, either B allows A to write, or A exits the **repeat** loop. □

Lemma 5.5 *After executing its first action, no process can perform a wrong writing.*

Proof. Process A can perform at most one *wrong* writing, and it may only happen when initially its program counter is set up after reading from the *Write* register and before writing in the *Read* register. Once this write action is executed, each write action of A in a *Read* register is performed within the context of an update. □

Lemma 5.6 *Let A and B be two neighbouring processes. After B executes its first action, if B allows A to write, then only the writing of A in its register $Write_{AB}$ may be able to cancel that permission.*

Proof. Nothing but writing into the register $Read_{BA}$ or into the register $Write_{AB}$ can cancel the writing permission. After B executes its first action, from Lemma 5.5 there is no *wrong* writing anymore. Hence, any writing into the register $Read_{BA}$ is executed within the context of a register's update. This update is such that the permission remains given to A, unless A writes into its register $Read_{BA}$ during the updating process or after the last update. □

Theorem 5.1 *Let A be a process. Whatever the execution, the system of processes which performs the protocol reaches a configuration in which A is not within the* **repeat** *loop anymore.*

Proof. Suppose A remains within the **repeat** loop forever; then A never writes into its *Write* registers. Every $4N_A$ actions, A is checking out the loop exiting condition. Whatever the execution, process A performs an infinite number of actions. Hence, A checks out the **repeat** loop exiting condition an infinite number of times. In particular, A tests the exit condition an infinite number of times after all its neighbours have already executed an action.

If at some test all neighbours of A allow its writing, then, at the next test, all its neighbours keep on giving A permission to write (by Lemma 5.6). In the meanwhile, A has updated its variables r_i and s_i, and when the test happens, the loop exiting condition is satisfied: A exits the loop.

Process A stays within the loop infinitely long in the case when, at each test, at least one neighbour does not allow its writing. Once a neighbour has allowed A to write, this neighbour cannot withdraw permission from A. Therefore, there exists at least one neighbour of A which never allows A to write. Now from Lemma 5.4, this is impossible, and the theorem follows. Therefore, the protocol is deadlock-free. □

Corollary 5.1 *Let A be a process. Whatever the execution, A writes an infinite number of times into all its Write registers.*

Proof. If A is out of the loop, then it takes A less than N_A actions to enter the loop. When it is within the **repeat** loop, then by Theorem 5.1, A cannot stay infinitely long. N_A actions after exiting the loop, A writes into all its *Write* registers and reenters the **repeat** loop. □

5.2 Correctness Proof of the Read Checking Protocol

Theorem 5.2 *Let A and B be two neighbouring processes. After B executes its first action and after any writing in the register $Write_{AB}$, A can write in the register $Write_{AB}$ only if B allows it, i.e. $Read_{BA} = Write_{AB}$ (see Definition 5.1).*

Proof. Process B is the ith neighbour of A. Between each of its two writings, A enters the **repeat** loop and exits the loop. Once A is within the loop, the register $Write_{AB}$ does not change value. The **repeat** loop's code is such that when the loop is exited, the value of the local variable s_i of A and the value of the register $Write_{AB}$ are equal. In the loop, the local variable r_i of A takes the value of the register $Read_{AB}$. The value of the register $Read_{BA}$ may change after this assignment and before the loop is exited. Thus, when the loop is exited two distinct cases have to be considered:

- No update of the register $Read_{BA}$ happens between the reading from that register and the loop exit. Then, $s_i = Write_{AB} = val_i = Read_{BA}$, and B allows the writing of A.
- Writings into the register $Read_{BA}$ happen between the reading from that register and the loop exit. However, the latter writings are performed within the context of updating. Hence, each time the value has changed, we have that $Read_{BA} = Write_{AB}$ and, by Lemma 5.6, the equality holds while A does not rewrite into the register $Write_{AB}$. □

After the writing of a value in the register $Write_{AB}$, the first primitive guarantees that A will only write in the register $Write_{AB}$ if B allows it. In the case when the value is new, B must perform the action **read**($Write_{AB}$) to allow the writing.

Summing up of the Results

1. **The protocol is live:** every process is updating all its *Write* registers an infinite number of times.
2. **The protocol is correct:** no process can write distinct values twice in a row in its *Write* register without any previous reading from that register.

6 Weak Rendezvous Protocol

In this section, we present a self-stabilizing *weak rendezvous* communications primitive.

Recall that The *rendezvous* mechanism (as defined in [16]) synchronizes communication in the link-register asynchronous model of distributed system: each *write* or *read* operation is performed in and from the same register. When Process A writes a value in its register $Write_{AB}$, it cannot perform *any other action* until process B has completed a *read* operation from the register $Write_{AB}$.

The *weak rendezvous* mechanism only requires that between two *write* operations performed by a process A in $Write_{AB}$, process B performs at least one *read* operation from $Write_{AB}$. Therefore, if A writes a value n consecutive times (even the same ones in each row) in the register $Write_{AB}$, the primitive guarantees that B copies each of the n values at least one time, once the system is stabilized.

The weak rendezvous mechanism is based upon the alternating bit technique. After writing in its register $Write_{AB}$, process A changes the value of the bit-register $Control_{AB}$. A can write again in the register $Write_{AB}$ only after B has copied the new value of $Control_{AB}$ into the register $CheckControl_{BA}$. And B copies the value only after reading in the register $Write_{AB}$.

The liveness proof of the weak rendezvous protocol is similar to the proof of the read checking protocol. The following Theorem 6.1 proves the correctness of the weak rendezvous protocol.

Theorem 6.1 *Let A and B be two neighbouring processes. After B executes its first action and after the xth (≥ 2) writing in the register $Write_{AB}$, B reads the value from $Write_{AB}$ before the next writing in $Write_{AB}$.*

Proof. As shown in Theorem 5.2, we can establish that before the xth writing in the register $Write_{AB}$, $Control_{AB} = CheckControl_{BA}$. After the writing in the register $Write_{AB}$, A changes the value in $Control_{AB}$ and enters the **repeat** loop ($Control_{AB} \neq CheckControl_{BA}$). A stays within the loop as long as B does not copy the value of $Control_{AB}$ into the register $CheckControl_{BA}$. Finally, B copies the value only after reading in the register $Write_{AB}$. □

The weak rendezvous protocol maintains a weak scheduling of the communication between processes in the following sense. We call a *weak scheduling* of the communication between process A and all its N_A neighbours the property that A can write twice into its registers $Write_{AB_i}$, only whenever all the B_i's did read from the register $Write_{AB_i}$ in the meantime ($1 \leq i \leq N_A$).

```
constant $N_A$   : the number of neighbours of $A$ ;
var  $r_i$       : message sent from the $i$th neighbour of $A$ ;
     $b_i$       : alternate bit sent from the $i$th neighbour of $A$ ;
     $c_i$       : alternate bit sent from A to the $i$th neighbour of $A$ ;
     $l_i$       : value of the last alternate bit sent from $A$ and read by the $i$th neighbour of $A$;
```

```
while true do
    for $i = 1$ to $N_A$ do
            write($Write_{AB_i}, get_i$) ;
            $c_i \leftarrow$ read($Control_{AB_i}$) ;
            write($Control_{AB_i}, (c_i + 1) \bmod 2$) ;
    endfor
    repeat
        for $i = 1$ to $N_A$ do
            $r_i \leftarrow$ read($Write_{B_iA}$) ;
            $b_i \leftarrow$ read($Control_{B_iA}$) ;
            write($CheckControl_{AB_i}, b_i$) ;
            $c_i \leftarrow$ read($Control_{AB_i}$) ;
            $l_i \leftarrow$ read($CheckControl_{B_iA}$) ;
        endfor
    until ( $\forall i \in [1, N_A]$  $c_i = l_i$ )
endwhile
```

Fig. 3. *The weak rendezvous protocol for A.*

7 Concluding Remarks

The paper presents two very basic general protocols for the design of fair and reliable self-stabilizing communication primitives. Both protocols work in arbitrary networks and also ensure minimal scheduling properties, whatever the initial configuration of the system of processes and the activations by the scheduler.

Each primitive can be used as a "black box" by a separate protocol, handling the procedures in more involved self-stabilizing algorithms. Thus, the protocols may be modified according to the designer's will and needs: e.g., in specific topologies of networks a weak scheduling of communications may impose fewer neighbours to read from the registers. For example, with only one neighbour, a point to point self-stabilizing pseudo-rendezvous mechanism may be completed. Along the same lines, the protocols also simulate reliable self-stabilizing message-passing in asynchronous distributed systems.

Although the paper does not concern itself with complexity measures, it is worth mentioning that when time is measured by some appropriately defined round complexity, the stabilization time of the read checking protocol is $O(1)$.

References

[1] **Y. Afek, G.M. Brown,** Self-Stabilization of the Alternating-Bit Protocol, *in the Proc. of the Symposium on Reliable Distributed Systems,* (1989) 80-83.

[2] Y. Afek, S. Kutten, M. Yung, Memory-efficient self-stabilization on general networks, *in the Proc of the 4th International Workshop on Distributed Algorithms and Graphs (WDAG'90), LNCS 486,* (Springer-Verlag 1990) 15-28.

[3] L.O. Alima, J. Beauquier, A.K. Datta, S. Tixeuil, Self-stabilization with global rooted synchronizers, *in the Proc. of the 18th International Conference on Distributed Computing Systems,* (1998) 102-109.

[4] E. Anagnostou, V. Hadzilacos, Tolerating Transcientand Permanent Failures, *in Proc. of the 7th Int. Workshop on Distributed Algorithms (WDAG'93), LNCS 725,* (Springer-Verlag 1993) 174-188.

[5] A. Arora, M.G. Gouda, Distributed reset, *IEEE Transactions on Computers*, vol. 43 (1994) 1026-1038.

[6] B. Awerbuch, Complexity of network synchronization, *J. of the Association for Computing Machinery*, vol. 32, No. 4 (1985) 804-823.

[7] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, Time optimal self-stabilizing synchronization, *in the Proc. of the 25th Annual ACM Symposium on Theory of Computing*, (1993) 652-661.

[8] B. Awerbuch, B. Patt-Shamir, G. Varghese, Self-Stabilization by Local Checking and Correction, *in the Proc. of the 31st Annual IEEE Symposium on Foundation of Computer Science*, (1991) 268-277.

[9] B. Awerbuch, G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols, *in the Proc. of the 31st Annual IEEE Symposium on Foundations of Computer Science*, (1991) 258-267.

[10] J. Beauquier, M. Gradinariu, C. Johnen, Memory space requirements for self-stabilizing leader election protocols, *in Proc. of the 18th Annual ACM Symposium on Principles of Distributed Computing*, (1999) 199-208.

[11] A. Bui, A.K. Datta, F. Petit, V. Villain, Space optimal and fast self-stabilizing pif in tree networks, *Technical Report RR. 98-06*, LaRIA, Université de Picadie (1998).

[12] S. Dolev, A. Israeli, S. Moran, Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity, *Distributed Computing*, 7 (1993) 3-16.

[13] S. Dolev, A. Israeli, S. Moran, Uniform dynamic self-stabilizing leader election, *IEEE Transactions on Parallel and Distributed Systems*, 8:4 (1997) 424-440.

[14] S. Dolev, M.G. Gouda, M. Schneider, Memory requirements for silent stabilization, *in Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing*, (1996) 27-34.

[15] M.G. Gouda, N. Multari, Stabilizing Communication Protocols, *IEEE Transactions on Computers*, 40 (1991) 448-458.

[16] C.A.R. Hoare, Communicating Sequential Processes, *Communication of the ACM*, vol. 21, No 8 (1978) 666-677.

[17] S.T. Huang, N.S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, *Information Processing Letters*, 41, 1992, 109-117.

[18] C. Johnen, L.O. Alima, A.K. Datta, S. Tixeuil, Self-stabilizing neighborhood synchronizer in tree networks, *in Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, 1999.

[19] I. Lavallée, C. Lavault, C. Johnen, Exorcisme ou communication fiable et équitable autostabilisée, *RR. 001, LRIA*, Université Paris 8 (Jan. 1998).

[20] G Varghese, Self-stabilization by counter flushing, *in Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing*, (1994) 244-253.