A Self-Stabilizing Communication Primitive

Colette Johnen, Ivan Lavallee, Christian Lavault

A Self-Stabilizing Communication Primitive

(Extended Abstract)

C. Johnen [*] I. Lavallée [†] C. Lavault [‡] [*]

[*] *LRI-CNRS Université Paris-Sud.*
[†] *LRIA-Paradis, Université Paris 8.*
[‡] *LIPN, CNRS UPRES-A 7030, Université Paris-Nord.*

Abstract

The goal of the paper is to provide designers of distributed self- stabilizing protocols with a fair and reliable communication primitive which allows any process which writes a value in its own registers to make sure that every neighbour eventually does read that value. 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 primitive runs a self-stabilizing protocol which implements a "rendezvous" communication mechanism in the link-register asynchronous model. This protocol works in arbitrary networks and also solves the problem of how to simulate reliable self-stabilizing message-passing in asynchronous distributed systems.

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

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). In this paper (see also [6]), we present a fair and reliable self-stabilizing communication primitive. It allows any process which writes a value in its own register(s) to make sure that every neighbour shall read that latter value, whatever the initial scheduling of processes' actions. Our communication primitive runs a self-stabilizing protocol on dynamic connected networks of arbitrary topology. Communication among neighbouring processes is carried out by the use of *communication registers* (called *registers* throughout the paper). The atomic operations that these

*Corresponding author: Christian Lavault LIPN, CNRS UPRES-A 7030, Université Paris-Nord, Av. J.-B. Clément 93430 Villetaneuse, France. Email: lavault@lipn.ura1507.univ-paris13.fr

registers support are *read* and *write*. This protocol implements various self-stabilizing variants of the *rendezvous* communication mechanism (as defined in [5]) and also allows simulation of self-stabilizing reliable message-passing in the link-register asynchronous model of distributed systems. Incidentally, it also maintains a weak scheduling of the communications between processes in arbitrary networks. Such a general primitive may prove useful as a basic communication tool for designing distributed self-stabilizing protocols.

Although distinct from the one described in [3], our model relies on close requirements and assumptions, especially in terms of communication (e.g., link registers, read/write atomicity, etc.). Several related communication problems in various self-stabilization settings (in the link-register or the message-passing model) have been addressed in the recent literature. A self-stabilizing communication protocol for two-way handshake is presented in [4], and a self-stabilizing version of the alternating-bit protocol is given in [1]. Though it is much closer, the problem addressed in [2] appears to meet less requirements than the solution given in the present paper, since the primitive does not ensure starvation-freeness. In [1], a process can prevent a neighbour from communicating new data, whereas our primitive is starvation-free. In that sense, our assumptions are safer and the problems considered are not equivalent.

Section 2 describes our model and gives the basic connected 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 and the corresponding self-stabilizing protocol is presented in Section 4. Section 5 is devoted to the proof of liveness and correctness of the protocol. Finally, the paper ends with few concluding remarks.

2 Model and Requirements

We model distributed self-stabilizing systems as a set of (possibly infinite) state machines called processes. Each process can communicate with some subset of the processes called its neighbours. We assume a *link-register* communication model under read/write atomicity [3]. Each link between any two neighbours A and B is composed of two pairs of registers[1], denoted $(Write_{AB}, Read_{AB})$ (belonging to A) and $(Write_{BA}, Read_{BA})$ (belonging to 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 can write in its own registers, $Write_{AB}$ and $Read_{AB}$, to communicate with B.

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

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

and distributed schedulers/demons are "equivalent", in the sense that any execution performed under a distributed scheduler may be simulated by a central one. In terms of communication, executions remain independent from any scheduling of processes' actions, either they are working in parallel or serially.

Note that self-stabilizing protocols offers full and automatic protection against all transient process failures, no matter how much the data have been corrupted; all values (constants, variables, etc.) within the registers may be fully corrupted. So, whatever the registers values, the protocol must secure the transfer of information between any two pair of neighbours after a "certain delay time". According to the specification of the protocol, it ensures that no process A can write twice in a row in its own registers $Write_{A-}$, without any previous reading from that register by either (at least) one neighbour or all its neighbours.

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.) Similarly, 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.*

Although the distributed system is asynchronous, the protocol actually implements a rendezvous communication primitive between A and B. When A sends a message to B, A becomes locked (A cannot exit the **repeat** loop) until B gets ready to receive the message.

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.

The 2-processes protocol implements a self-stabilizing two-way rendezvous mechanism and, therefore, allows to simulate a self-stabilizing protocol in the message passing communication model.

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.

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 General Self-Stabilizing 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 protocol in arbitrary networks for a process A, with neighbours B_i's ($1 \leq i \leq N_A$), is as follows.

```
constant  N_A   : the number of neighbours of A ;
var  s_i        : message to be sent to the ith neighbour of A ;
     r_i        : message sent from the ith neighbour of A ;
     val_i      : value of the last message sent from A and read by the ith 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 ← read(Write_{B_iA}) ;
         write(Read_{AB_i}, r_i) ;
         val_i ← read(Read_{B_iA}) ;
         s_i ← read(Write_{AB_i}) ;
     endfor
   until ( ∀i ∈ [1, N_A]  val_i = s_i )
endwhile
```

Fig. 2. *The general n-processes protocol for A.*

5 Proving Properties of the General Protocol

5.1 Proof of Liveness

Lemma 5.1 *Let γ be any configuration of an arbitrary network of processes on which the general protocol is performed. Then no process deadlocks 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. □

The following Lemma 5.2 and Lemma 5.3 are immediate consequences of Lemma 5.1.

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

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

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

Lemma 5.5 and Lemma 5.6 derive easily from Lemma 5.4 and the above definitions.

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

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.*

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.*

5.2 Correctness Proof of the Protocol

Definition 5.3 *Process B is said to read the value of process A iff $Read_{BA} = Write_{AB}$.*

Theorem 5.2 *Let A and B be two neighbouring processes. After B executes its first action and after any writing in the register $Write_{AB}$, B reads the value of A before a next writing in the register $Write_{AB}$.*

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 did read the value 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 this register. 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 $Write_{AB}$. □

Summing up of the results

First, the protocol is deadlock-free, since every process is updating all its *Write* registers an infinite number of times. Second, the protocol is correct, since no process can write twice in a row in its *Write* register without any previous reading from that register by (at least) one neighbour.

Remark: Let A be a process and let A_i denote any of its N_A neighbours ($1 \leq i \leq N_A$).

From Subsection 5.2, the notion of *weak scheduling* of the communication between processes is easily derived. We call a *weak scheduling* of the communication between process A and all its

neighbours the property that A cannot write twice into its registers $Write_{AA_i}$ but only whenever all the A_i's did read from the register $Write_{AA_i}$ in the meantime.

Therefore, from Theorem 5.2, the general protocol maintains a weak scheduling of the communication between processes in the above sense.

6 Concluding Remarks

We presented a basic and general protocol for the design of a fair and reliable self-stabilizing communication primitive. This self-stabilizing distributed protocol implements a self-stabilizing version of the well-known rendezvous communication primitive (as defined by Hoare in [5]) in the link-register asynchronous model of distributed system. It works in arbitrary networks and also ensures minimal scheduling properties, whatever the initial configuration of the system of processes and their activations by the scheduler.

Besides, the protocol 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 rendezvous mechanism may be completed. Along the same lines, the protocol also simulates 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 general protocol is $O(1)$.

References

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

[2] **E. Anagnostou, V. Hadzilacos.** Tolerating Transcient and Permanent Failures. *in Proc. of the 7th Int. Workshop on Distributed Algorithms (WDAG93), LNCS 725*, Springer-Verlag, 1993, 174-188.

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

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

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

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