Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Towards self-stabilizing wait-free shared memory objects

J.H. Hoepman, M. Papatriantafilou and P. Tsigas

Computer Science/Department of Algorithmics and Architecture

CS-R9514 1995

Report CS-R9514
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Towards Self-Stabilizing Wait-Free Shared Memory Objects

Jaap-Henk Hoepman
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
jhh@cwi.nl

Marina Papatriantafilou
Max Planck Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
ptrianta@cti.gr

Philippas Tsigas
Max Planck Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
tsigas@cti.gr

Abstract

Past research on fault tolerant distributed systems has focussed on either processor failures, ranging from benign crash failures to the malicious byzantine failure types, or on transient memory failures, which can suddenly corrupt the state of the system. An interesting question in the theory of distributed computing is whether one can device highly fault tolerant protocols which can tolerate both processor failures as well as transient errors. To answer this question we consider the construction of self-stabilizing wait-free shared memory objects. These objects occur naturally in distributed systems in which both processors and memory may be faulty. Our contribution in this paper is threefold. First, we propose a general definition of a self-stabilizing wait-free shared memory object that expresses safety guarantees even in the face of processor failures. Second, we show that within this framework one cannot construct a self-stabilizing single-reader single-writer regular bit from single-reader single-writer safe bits. This result leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit with which, as a third contribution, we construct self-stabilizing regular and atomic registers.

CR Subject Classification (1991): B.3.2, B.4.3, B.4.5, C.1.2, C.2.4, C.4, D.1.3, D.4.1, D.4.5
Keywords & Phrases: Distributed Systems, Shared Memory, Communication, Synchronization, Self-Stabilization, Fault-Tolerance, Wait-Freedom, Processor Crashes, Memory Failures.
Note: The first and third author were partially supported by the Dutch foundation for scientific research (NWO) through NFI Project ALADDIN, under contract number NF 62-376. The second author was partially supported by a NUFFIC Fellowship. This work was conceived when the second and third author were visiting the CWI.

1. Introduction

The importance of reliable distributed systems can hardly be exaggerated. In the past, research on fault tolerant distributed systems has focused either on system models in which only processors fail, or on system models in which only the memory is faulty. In the first model a distributed system must remain operational while a certain fraction of the processors is malfunctioning. In the construction of shared memory objects like atomic registers, this issue is addressed by considering *wait-free* constructions which guarantee that any operation executed by a single processor is able to complete even if all other processors crash in the meantime. Originally, research in this area focussed on the construction of atomic registers from weaker (i.e. safe or regular) ones [VA86, Lam86, PB87, LTV89, IS92]. Later attention shifted to stronger objects [AH90, Her91]. See [KK89] for a brief survey.

In the second model a distributed system is required to overcome arbitrary changes to its state within a bounded amount of time. If the system is able to do so, it is called *self-stabilizing*. Self-stabilizing protocols have been extensively studied in the past. Originating from the work of Dijkstra on self-stabilizing mutual exclusion on rings [Dij74], several other self-stabilizing protocols have been proposed for particular problems, like mutual exclusion on other topologies [BGW89, BP89, DIM93], the construction of a spanning-tree [AKY90], orienting a ring [IJ93, Hoe94], and network synchronization [AKM+93]. Another approach focusses on the construction of a 'compiler' to automatically transform a protocol belonging to a certain class to a similar, self-stabilizing, one [KP90, AKM+93]. For a survey on self-stabilization, see [Sch93].

To develop truly reliable systems both failure models must be considered together. We briefly summarize recent theoretical research taken into this direction. Anagnostou and Hadzilacos [AH93] show that no self-stabilizing, fault-tolerant, protocol exists to determine, even approximately, the size of a ring. Gopal and Perry [GP93] present a 'compiler' to turn a fault-tolerant protocol for the synchronous rounds message-passing model into a protocol for the same model which is both fault-tolerant and self-stabilizing. A combination of self-stabilization and wait-freedom in the construction of fault-tolerant clock-synchronization protocols is presented in [DW93, PT94].

In this paper we consider the construction of self-stabilizing wait-free shared memory objects in asynchronous distributed systems. A shared memory object is a data structure stored in shared memory which may be accessed concurrently by several processors in the system through the invocation of operations defined for the object. Such objects occur naturally in distributed systems in which both processors and memory may be faulty. We give a general definition of self-stabilizing wait-free shared memory objects, and focus on the construction of several types of self-stabilizing wait-free shared registers. Single writer single reader safe bits—traditionally used as the weakest building block in such constructions—are shown to be too weak for our purposes.

Shared registers are shared objects reminiscent of ordinary variables, but which can be read or written by different processors concurrently. They are distinguished by the level of consistency guaranteed in the presence of concurrent operations [Lam86]. A register is *safe* if a read returns the most recently written value, unless the read is concurrent with a write in which case it may return an arbitrary value. A register is *regular* if a read returns the value

written by concurrent or immediately preceding write. A register is *atomic* if all operations on the register appear to take effect instantaneously. Shared registers are also distinguished by the number of processors that may invoke a read or a write operation, and by the number of values it may assume. These dimensions imply a hierarchy with single-writer single-reader ($1W1R$) binary safe registers (a.k.a. bits) on the lowest level, and multi-writer multi-reader ($nWnR$) l-ary atomic registers on the highest level. A *construction* of a register is comprised of i) a data structure consisting of memory cells called *sub-registers* and ii) a set of read and write procedures which provide the means to access it. Upon invocation each procedure performs sequential steps, which may be either read/write *sub-operations* on the sub-registers or some local computations.

Li and Vitányi [LV92] and Israeli and Shaham [IS92] where the first to consider self-stabilization in the context of shared memory constructions. They implicitly call a shared memory construction self-stabilizing if for every *fair* execution started in an arbitrary state, the object behaves according to its specification except for a finite prefix of the execution. We feel, however, that this notion of a self-stabilizing object does not agree well with the additional requirement that the object is wait-free, since self-stabilization of the object now only guarantees recovery from transient errors in fair executions (in which no processors crash), while an object should be wait-free to ensure that a single processor can make progress even if all other processors have crashed (cf. section 2). Our contribution in this paper is threefold. First, in section 2, we propose a general definition of a self-stabilizing wait-free shared memory object, that ensures that all operations after a transient error will eventually behave according to their specification even in the face of processor failures. Second, in section 3, we show that within this framework one cannot construct a self-stabilizing single-reader single-writer regular bit from single-reader single-writer safe bits. This result leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit. These bits are then, as a third contribution, used to construct a self-stabilizing single-writer single reader regular bit in section 4, a self-stabilizing l-ary single-writer single-reader regular register in section 5, and a self-stabilizing l-ary multi-writer multi-reader atomic register in section 6. Section 7 concludes this paper with directions for further research.

2. Defining self-stabilizing wait-free objects

In the definition of shared memory objects we follow the concept of *linearizability* ([Her91]) which we, for the sake of self containment, briefly rephrase here. Consider a distributed system of n sequential processors. A shared memory object is a data-structure stored in shared memory which may be accessed by several processors concurrently. Such an object defines a set of *operations* $\mathcal{O}$ which provide the only means for a processor to modify or inquire the state of the object. The set of processors that can invoke a certain operation may be restricted. Each operation $O \in \mathcal{O}$ takes zero or more parameters p on its invocation and returns a value r as its response. Each such operation execution is called an *action*, denoted by $r = O(p)$. Actions take some time to complete, hence there is a time interval between the invocation of an action and its corresponding response. As processors are sequential, they cannot invoke an action if their previously invoked action is still *pending*, i.e. has not responded yet.

The desired behaviour of the object is described by its *sequential specification* $\mathcal{S}$. This

specifies the state of the object, and for each operation its effect on the state of the object and its response. We write $(s, r = O(p), s') \in \mathcal{S}$ if invoking O with parameters p in state s changes the state of the object to s' and returns r as its response.

A *run* over the object is a tuple $\langle \mathcal{A}, \rightarrow \rangle$ with actions $\mathcal{A}$ and partial order $\rightarrow$ such that for $A, B \in \mathcal{A}$, $A \rightarrow B$ iff the response for A occurred before the invocation of B. Runs have infinite length and capture the real time ordering between actions invoked by the processors. A *sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$ over the object is an infinite sequence $s_0 A_0 s_1 A_1 \ldots$, where $\bigcup_i A_i = \mathcal{A}$, s_i a state of the object as in its sequential specification, and $\Rightarrow$ a total order over $\mathcal{A}$ defined by $A_i \Rightarrow A_j$ if and only if $i < j$. A run $\langle \mathcal{A}, \rightarrow \rangle$ corresponds with a sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ if the set of actions $\mathcal{A}$ is the same in both, and if $\Rightarrow$ is a total extension of $\rightarrow$ (i.e. $A \rightarrow B$ implies $A \Rightarrow B$). Stated differently, the sequential execution corresponding to a run is a run in which no two actions are concurrent but in which the 'observable' order of actions in the run is preserved.

Definition 1 *A run $\langle \mathcal{A}, \rightarrow \rangle$ is* linearizable w.r.t. sequential specification $\mathcal{S}$*, if there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$, such that $\Rightarrow$ is an extension of $\rightarrow$, and for all i we have $(s_i, A_i, s_{i+1}) \in \mathcal{S}$.*

An object is linearizable w.r.t. its sequential specification $\mathcal{S}$ if all possible runs over the object are linearizable w.r.t. $\mathcal{S}$. Informally speaking, an object is linearizable w.r.t. to specification $\mathcal{S}$ if all actions appear to take effect instantaneously and act according to $\mathcal{S}$.

2.1 Adding self-stabilization

Li and Vitányi [LV92] and Israeli and Shaham [IS92] are the first to consider self-stabilizing wait-free constructions. Both papers implicitly use the following straightforward definition of a self-stabilizing wait-free object.

Definition 2 *A shared wait-free object is self-stabilizing if an arbitrary* fair *execution (in which all operations on all processors are executed infinitely often) started in an arbitrary state, is linearizable except for a finite prefix.*

A moment of reflection shows that assuming fairness may not be very reasonable for wait-free shared objects. The above definition requires that after a transient error all processors have to cooperate to repair the fault. But wait-freedom was introduced to ensure that processors could make sensible progress even if other processors had crashed. Moreover, the above definition does not give a bound on the length of the finite prefix, and thus does not give a bound on the stabilization time. These observations lead us to the following stronger, informal, definition of a self-stabilizing shared object.

Definition 3 *A shared wait-free object is self-stabilizing, if an arbitrary execution started in an arbitrary state is linearizable except for a bounded finite prefix.*

For the formal definition that is to follow, let us define $\mathsf{count}(A)$ equal to i if A is executed as the i-th action of a certain processor, where $\mathsf{count}(A) = 1$ for all first actions of the processors. As actions are unique, this is well-defined.

Definition 4 *A run* $\langle \mathcal{A}, \rightarrow \rangle$ *is* linearizable w.r.t. sequential specification $\mathcal{S}$ after k processor actions*, if there exists a corresponding sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$*, such that* $\Rightarrow$ *is an extension of* $\rightarrow$*, and for all* i*, if* $\mathsf{count}(A_i) > k$ *then* $(s_i, A_i, s_{i+1}) \in \mathcal{S}$*.*

Here k is called the delay of the implementation. Note that the definition allows the first k actions of a processor to behave arbitrarily (even so far as to allow e.g. a read action to behave as a write action or vice versa), but the effect of such an arbitrary action should be globally consistent. Thus this definition gives the strong guarantee that all actions following such arbitrarily behaving actions will agree on how that action actually behaved. In particular, for $k = 0$, the definition implies that all actions agree on the effect of the transient error on the state of the object. For example, for a shared register all reads that occur immediately after a transient error should return the same value.

Definition 5 *An implementation of a shared object specified by sequential specification* $\mathcal{S}$ *is called* k self-stabilizing wait-free *if all its runs starting in an arbitrary initial state are linearizable w.r.t.* $\mathcal{S}$ *after* k *processor actions, and all actions complete within a bounded number of steps.*

In the above definition the delay k of the implementation is taken to be independent of the type of operations performed by a processor, while one might very well feel that the difficulty of stabilizing different types of operations on the same object may vary. Indeed, preliminary versions of this definition were more fine-grained and included separate delays for different types of operations (e.g. allowing the first k_w writes and the first k_r reads performed by a processor on a read/write register to be arbitrary). While in general it is unclear whether this amount of detail is really necessary, it turned out that for our present purposes it is not. It seems that different operations on a shared memory object already need to reach agreement on the state of the object, irrespective of the actual type of operation.

In this paper we build the foundations for analyzing and implementing self-stabilizing wait-free shared memory objects. Apart from giving a general definition of such objects, we want to show that their construction uses similar methods as those used in classical, non-stabilizing, objects. The main theme in the remainder of this paper is to give a construction of an atomic, linearizable, shared variable from weaker types of variables identified in the literature. These variables are distinguished by the level of consistency guaranteed in the presence of concurrent operations [Lam86]. Giving self-stabilizing constructions of stronger types of variables from weaker ones shows that the level of synchronization required for these constructions can indeed be implemented in the face of both processor and memory failures. Then in what follows we also need the definition of safe and regular self-stabilizing registers. Let us write $W \rightleftarrows R$ if for a write W, we have that $W \rightarrow R$ and that there is no write W' such that $W' \rightarrow R$. If no such write exists, we take the imaginary initial write $W_\perp$ responsible for writing the arbitrary initial value. Let us write $W \| R$ if neither $W \rightarrow R$ nor $R \rightarrow W$. Define the *feasible* writes of a read R as all writes W such that $W \rightleftarrows R$ or $W \| R$.

Definition 6 *A register is* k*-stabilizing wait-free if for all its runs starting in an arbitrary initial state only the first* k *actions performed by one processor behave arbitrarily after the error, and all actions finish within a bounded number of steps. Such a register is safe if for*

all other read actions R not concurrent with a write action, R returns the value written by W with $W \rightleftarrows R$. Such a register is regular if all other read actions return the value written by a feasible write.

3. Why 0-stabilizing $1W1R$ safe bits are not strong enough.

In this section we prove that there exists no implementation of a wait-free k-stabilizing regular register using 0-stabilizing $1W1R$ binary safe sub-registers.

It is a common convention to view the schedule as being chosen by an *adversary*, who seeks to force the protocol to behave incorrectly. This descriptive tool is used throughout this section. The adversary is in control of (i) choosing the configuration of the system after a transient error and (ii) scheduling the process steps in a run.

The heart of the problem of such an implementation can be described as follows: Since the writer (the reader) does not have read access to the sub-registers which it can write, in order to know their contents and settle itself into correct stabilized behaviors, it has to rely on information that either is local or is passed to it though shared sub-registers that can be written only by the reader (the writer, respectively). But then the adversary can set the system in a state in which this information is inconsistent. Subsequently, by scheduling the processes' sub-actions on the same sub-register to be concurrent, it can destroy the information propagation because of the weak consistency that the safe sub-registers guarantee. In this way it misleads the processes into incorrect behaviors.

Now suppose, for the sake of argument, that there exists a wait-free k-stabilizing implementation of an $1W1R$ binary regular register from 0-stabilizing $1W1R$ binary safe sub-registers. Let S_R, S_W denote the reader and writer process' local state, respectively. Such a construction must use two sets of binary sub-registers (that can be considered as two "big" sub-registers): one that can be written by the writer and read by the reader and one that can be written by the reader and read by the writer. Let them be denoted by Reg_{WR} and Reg_{RD}, respectively. A system configuration C is a tuple $(S_R, S_W, Reg_{RD}, Reg_{WR})$ with values for the processes' local states and shared sub-registers.

A read on the regular register may involve several sub-reads of Reg_{WR}. However, in the course for a contradiction, attention may be restricted to executions in which each of those sub-reads obtains the same value of Reg_{WR}, say rw. Then, the value returned by the read is determined by a *reader function* $f_R(sr, rw)$, with input parameters the reader's local state sr and the value rw.

Theorem 7 *There exists no deterministic implementation of a wait-free k-stabilizing $1W1R$ binary regular register using $1W1R$ binary 0-stabilizing safe sub-registers.*

Proof: Towards a contradiction, suppose that such an implementation exists.

Let rw be one of the values that Reg_{WR} can hold. We say that a state sr of the reader is *settled* w.r.t. rw if for any read action R in any possible run starting in a configuration with $S_R = sr$ and $Reg_{WR} = rw$ it holds that R will return a value which was the input parameter of one of its feasible writes in the run. Now, let $SettlStat(rw)$ be the local states of the reader that are settled w.r.t. some rw that can be a value of Reg_{WR}. For any rw, $SettlStat(rw)$

is not empty. This is so because, from our assumption that such an implementation exists, starting a run from a configuration with $Reg_{WR} = rw$ such that the reader performs k reads while the writer performs no write action, the system will come in a configuration with the reader in a state that is settled w.r.t. rw. Moreover, for all the configurations in which $S_R = sr'$ and $Reg_{WR} = rw'$ that result from any continuation of the run with any possible interleaving of process steps, it will hold that sr' is settled w.r.t. rw'.

Let $C = (S_R = sr, S_W = sw, Reg_{RD} = rr, Reg_{WR} = rw_1)$ be a system configuration, such that sr is settled w.r.t. rw_1 and $f_R(sr, rw_1) = 1$. After a $Write(0)$ action W is scheduled starting from C, the system will be in a configuration $C' = (S_R = sr, S_W = sw', Reg_{RD} = rr, Reg_{WR} = rw_0)$. It must hold that sr is settled w.r.t. rw_0 and $f_R(sr, rw_0) = 0$, i.e. a read starting after the completion of W should return 0, since W is its only feasible write.

Now consider what the adversary can do:

(a) Let it set the system in $C_0 = (S_R = sr, S_W = sw, Reg_{RD} = rr, Reg_{WR} = rw_0)$ and schedule a $Write(1)$ action W'. The writer has to modify Reg_{WR} into rw'_1 such that $f_R(sr, rw'_1) = 1$. In order to do so, it performs sub-writes $s_1, \ldots, s_m$ on bits of Reg_{WR} in that order. We write $rw \setminus s_1..s_i$ to denote the value obtained after sub-writes $s_1, \ldots, s_i$ have taken place on Reg_{WR}, while Reg_{WR} held rw initially. There will be an s_i, with $1 \leq i \leq m$ and $rw' = rw_0 \setminus s_1..s_{i-1}$, $rw'' = rw_0 \setminus s_1..s_i$, such that $f_R(sr, rw') = 0$ while $f_R(sr, rw'') = 1$. The former value for Reg_{WR} could be obtained by a read scheduled to start immediately after the sub-write s_{i-1} and end before any other sub-operation of W'; the latter could be obtained by a read scheduled to start immediately after the sub-write s_i and end before any other sub-operation of W'. Clearly, if the value written by s_i is v ($\in \{0, 1\}$), the value of the respective bit in rw' is $\neg v$.

(b) Now let the adversary set the system in $C_1 = (S_R = sr, S_W = sw, Reg_{RD} = rr, Reg_{WR} = rw'')$ and schedule again a $Write(1)$ action. Again sr is settled w.r.t. rw'', since it is a result of an operation which modified Reg_{WR} from rw_0 (with respect to which sr was settled) into rw''. The writer does not know the contents of Reg_{WR}, hence, it performs the same sequence of sub-writes $s_1, \ldots, s_m$, as before. If the adversary schedules a read R to take place entirely in the duration of the sub-write s_i, then R might obtain either rw' or rw'' as values of Reg_{WR}, due to the fact that the sub-registers of the construction are safe. But now it should be $f_R(sr, rw') = f_R(sr, rw'') = 1$, since the implementation must be regular. But in (a) we had that $f_R(sr, rw') = 0$ and $f_R(sr, rw'') = 1$, a contradiction. ■

4. Constructing a 0-stabilizing $1W1R$ regular bit

In protocol 1 we present a construction of a 0-stabilizing $1W1R$ regular bit from a 0-stabilizing $1W2R$ safe bit. The protocol is similar to the original construction of a regular bit by Lamport ([Lam86], construction 3). He observed that for a binary register, assuming only the values 0 and 1, the only difference between a safe and a regular register is that if all feasible writes of a read operation write the same value a and at least one such write overlaps with the read, then for a safe bit the read may return either a or $\neg a$, whereas for a regular bit the read must return a. Then if the processor writing to the safe bit maintains a local copy of the last written value, and only writes a new value if this new value differs from the previous value stored in the register, then this actually simulates a regular register.

S: 0-stabilizing $1W2R$ `safe` bit

operation $Read(R) : \{0,1\}$
  return $(Read(S))$

operation $Write(R, v : \{0,1\})$
$l : \{0,1\}$
  $l := Read(S)$
  if $l \neq v$ **then** $Write(S, v)$

Protocol 1: 0-stabilizing $1W1R$ regular bit from 0-stabilizing $1W2R$ safe bit

As we have already shown in the previous section, this approach cannot work immediately for the implementation of a self-stabilizing regular bit. The core of the problem is that due to a transient error, the local copy of the last written value may be out of sync with the actual value present in the register. If we do not take this possibility into account, and do not really write a value to the safe bit once in a while, then the reader may read the wrong value forever. On the other hand, if we do occasionally write the safe register even though the local copy tells us this is unnecessary, then we may interfere with a read—thus returning an arbitrary value—whose feasible writes actually all wrote the same value.

The same idea can be applied, however, if we use a $1W2R$ safe bit and let the writer read this register instead of relying on a local copy of its value. Assuming such a $1W2R$ safe bit is not much stronger than assuming a $1W1R$ safe bit, as the latter models a flip-flop with a single output wire, whereas the first models a flip-flop with its output wire split in two. This idea is applied in protocol 1, whose correctness is stated in the next theorem.

Theorem 8 *Protocol 1 implements a wait-free 0-stabilizing $1W1R$ regular binary register using one 0-stabilizing $1W2R$ safe binary register.*

Proof: Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run over the regular bit; this induces a run $\langle \mathcal{A}', \rightarrow' \rangle$ over the safe bit S which is 0-stabilizing. Let $w_\perp$ be the initializing write of $\langle \mathcal{A}', \rightarrow' \rangle$; and choose $W_\perp$ to write the same value as $w_\perp$. According to definition 6 we have to show that in $\langle \mathcal{A}, \rightarrow \rangle$ all reads return the value written by a feasible write.

Now suppose a read RR returns the value a; as the register is binary this can only be wrong if all feasible writes write $\neg a$. Then the write WR with $WR \rightleftarrows RR$ writes $\neg a$, and all writes WR' concurrent with RR write $\neg a$ and we have $WR \rightarrow WR'$. Note that WR and WR' read S without interference from $\langle \mathcal{A}', \rightarrow' \rangle$. Now either WR reads $\neg a$ from S and hence does not write to S, or WR reads a from S and hence writes $\neg a$ to S, or WR is the imaginary initial write $W_\perp$ writing $\neg a$. In all these cases the first WR' with $WR \rightarrow WR'$ also reads $\neg a$ from S in $\langle \mathcal{A}', \rightarrow' \rangle$ and hence does not write S in $\langle \mathcal{A}', \rightarrow' \rangle$. Continuing this argument for all WR' concurrent with RR we see that none of them writes S. Then RR reads S without interference in $\langle \mathcal{A}', \rightarrow' \rangle$ and as $WR \rightleftarrows RR$, reads $\neg a$ from S and returns $\neg a$, a contradiction. ■

5. Constructing a 0-stabilizing $1W1R$ l-ary regular register

In protocol 2 we present a construction of a 0-stabilizing $1W1R$ l-ary regular register from l 0-stabilizing $1W1R$ regular bits. The protocol is similar to the original construction of a

$R_0 \ldots R_{l-1}$: 0-stabilizing $1W1R$ **regular** bit

operation $Write(R, v : \{0, \ldots, l-1\})$
  $Write(R_v, 1)$
  while $v \neq 0$ **do** $v := v - 1$; $Write(R_v, 0)$

operation $Read(R) : \{0, \ldots, l-1\}$
$w : \{0, \ldots, l\}$
  $w := 0$
  while $Read(R_w) = 0 \wedge w < l$
  do $w := w + 1$
  if $w = l$ **then return** $(l-1)$
    else return (w)

Protocol 2: 0-stabilizing $1W1R$ l-ary regular from 0-stabilizing $1W1R$ regular bits

regular multiple valued register by Lamport ([Lam86], construction 4), except that in our protocol we have to take into account that a transient error may cause all bits to be set to 0. In that case a default value $(l-1)$ is returned. Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run over the regular l-ary register; this induces a run $\langle \mathcal{A}', \rightarrow' \rangle$ over the regular bits R_i which are 0-stabilizing. Let $w_{i,\perp}$ be the initializing write of $\langle \mathcal{A}', \rightarrow' \rangle$ for register R_i; and choose $W_\perp$ to write the minimal v such that $w_{v,\perp}$ wrote 1, setting $v = l-1$ if no such v exists. Let us write RR_v for the read of R_v by read RR, and let us write $WR_v(a)$ for the write to R_v by a write WR. We first prove the following lemma

Lemma 9 *If $WR_i(1) \not\rightarrow' RR_i$ then RR_i returns 1 or there exists a $j > i$ and a $WR'(j)$ such that $WR'_j(1) \rightarrow' RR_{i+1}$. Moreover, if $WR_i(1) \rightarrow' RR_i$ then $WR(i) \rightarrow WR'(j)$.*

Proof: If RR_i does not return 1, there must be $WR'_i(0)$ feasible for RR_i. Then $WR'_i(0) \not\rightarrow' RR_i$ (and if $WR_i(1) \rightarrow' RR_i$ we must have $WR(i) \rightarrow WR'$). Then WR' must write a $j > i$ and we have $WR'_j(1) \rightarrow' WR'_i(0) \not\rightarrow' RR_i \rightarrow' RR_{i+1}$. Then $WR'_j(1) \rightarrow' RR_{i+1}$. ■

Theorem 10 *Protocol 2 implements a 0-stabilizing $1W1R$ l-ary regular register using l 0-stabilizing $1W1R$ regular binary registers.*

Proof: According to definition 6 we have to show that in $\langle \mathcal{A}, \rightarrow \rangle$ all reads return the value written by a feasible write. If for a read RR, for all i, RR_i returns 0, then it returns $l-1$. Applying lemma 9 inductively (starting at $i = l-1$) we conclude that for no i, there is a $WR_i(1)$ such that $WR_i(1) \not\rightarrow' RR_i$. Then there is no WR such that $WR \rightarrow RR$, except for $W_\perp$, and for all i the only feasible write for RR_i is $w_{i,\perp}$. Hence $W_\perp \rightleftarrows RR$ and for all i, $w_{i,\perp}$ wrote 0, so $W_\perp$ is feasible for RR writing $l-1$ by definition.

Now consider a read RR where RR_i returns 1 and for all $j < i$, RR_j returns 0; this read returns i. Because R_i is 0-stabilizing regular there must be a $WR(i)$ with subwrite $WR_i(1)$ such that $WR_i(1) \not\rightarrow' RR_i$ (or $WR(i) = W_\perp$) and hence $WR(i) \not\rightarrow RR$. The only case where $WR(i)$ would not be a feasible write for RR occurs if there exists a WR' such that $WR(i) \rightarrow WR'(j) \rightarrow RR$. If $j \geq i$, then WR' writes 0 or 1 to R_i by WR'_i with $WR_i \rightarrow WR'_i \rightarrow RR_i$ contradicting that WR_i was a feasible write for RR_i. If $j < i$, then by lemma 9 and the fact

that RR reads $R_j = 0$ we must have a $k > j$ (and $k < i$ by the above) and a $WR''(k)$ such that $WR'(j) \to WR''(k)$ and $WR'_k(1) \to' RR_{j+1}$ (and hence $WR'_k(1) \to' RR_k$). But then there must be k' with $k < k' < i$ and similar requirements. As this sequence stops at i, no such j can exist either. If $WR(i) = W_\perp$, the initializing write, then for all $j < i$ RR read the initial value 0 from R_j which implies that v indeed equals the initial value for the register. ■

6. Constructing a 1-stabilizing $nWnR$ l-ary atomic register

In protocol 3, we construct a 1-stabilizing $nWnR$ l-ary atomic register from $1R1W$ ∞-ary 0-stabilizing regular registers. This protocol is an adaption of the Vitányi and Awerbuch [VA86, AKKV88] multi-writer atomic register. The only difference is that here the maximum is taken over the triple of tags, identities and values, whereas in the original construction the maximum is taken over the tags and identities only.

$R_{11} \ldots R_{nn}$: 0-stabilizing $1W1R$ `regular`: $\mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
the fields *tag*, *id*, and *val*

operation $Write_i(R, v : \mathcal{V})$
$max : \mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
$max := \max_{1 \le j \le n} Read(R_{ji})$
for $j := 1$ **to** n
do $Write(R_{ij}, \langle max.tag + 1, i, v \rangle)$

operation $Read_i(R) : \mathcal{V}$
$max : \mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
$max := \max_{1 \le j \le n} Read(R_{ji})$
for $j := 1$ **to** n
do $Write(R_{ij}, max)$
return $(max.val)$

Protocol 3: 1-stabilizing $nWnR$ l-ary atomic from 0-stabilizing $1W1R$ ∞-ary regular

---

In the protocol, $\mathcal{V}$ is the domain of values written and read by the multi-writer register. The constructions uses n^2 regular 0-stabilizing regular registers R_{ij} written by processor i and read by processor j. These register store a *label* consisting of an unbounded *tag*, a processor *id* with values in the domain $\{1, \ldots, n\}$, and a *value* in $\mathcal{V}$. Labels are lexicographically ordered by $\le$.

6.1 Proof of correctness

Let $\langle \mathcal{A}, \to \rangle$ be a run of the above protocol. Define for $A \in \mathcal{A}$, $\mathsf{label}(A)$ as the label written by A, and $\mathsf{max}(A)$ as the maximal label among those read and stored in *max*. We closely follow the proof as given in [AKKV88].

Partition the set of actions $\mathcal{A}$ into reads $\mathcal{R}$ and writes $\mathcal{W}$. As actions A with $\mathsf{count}(A) = 1$ can behave arbitrary, we have to be careful about the class we put these actions in. Define

$$\begin{aligned} \mathcal{F} &= \{A \in \mathcal{A} \mid \mathsf{count}(A) = 1\} \\ \mathcal{R}^- &= \{A \in \mathcal{A} \mid \mathsf{count}(A) > 1 \text{ and } A \text{ is a read}\} \\ \mathcal{W}^- &= \{A \in \mathcal{A} \mid \mathsf{count}(A) > 1 \text{ and } A \text{ is a write}\} \end{aligned}$$

Now $\mathcal{F}$ has to be further subdivided into actions $\mathcal{F}_W$ that seem to behave as a write and

actions $\mathcal{F}_R$ that seem to behave as a read, as follows. This is a bit tricky because we want to make sure that we can give a nice definition of the reading function π later on. This requires that no two apparent writes can write the same label; hence we have added the processor identifier and the written value to the tag. For a set of actions $\mathcal{F}$, define $\mathsf{label}(\mathcal{F}) = \{\mathsf{label}(F) \mid F \in \mathcal{F}\}$. Now set $\mathcal{F}_l = \mathsf{label}(\mathcal{F}) \setminus \mathsf{label}(\mathcal{W}^-)$, the set of labels not written by a real write in $\mathcal{W}^-$. Define $\mathcal{F}_W$ as an arbitrary subset of F such that

(F1) For all $A \in \mathcal{F}_W$, $\mathsf{label}(A) \in \mathcal{F}_l$, and

(F2) $\mathsf{label}(\mathcal{F}_W) = \mathcal{F}_l$, and

(F3) For all $A, B \in \mathcal{F}_W$ with $\mathsf{label}(A) = \mathsf{label}(B)$ we have $A = B$, and

(F4) For all $A \in \mathcal{F}_W$ and $B \in \mathcal{F}$, if $\mathsf{label}(A) = \mathsf{label}(B)$ then $s(A) < s(B)$ (where $s(A)$ is the start time of A) so also $B \not\rightarrow A$.

Now set $\mathcal{F}_R = \mathcal{F} \setminus \mathcal{F}_W$ and define $\mathcal{W} = \mathcal{W}^- \cup \mathcal{F}_W$ and $\mathcal{R} = \mathcal{R}^- \cup \mathcal{F}_R$.

Lemma 11 *If $A \rightarrow B$ then $\mathsf{label}(A) \leq \mathsf{label}(B)$. If $B \in \mathcal{W}^-$ this inequality is strict.*

Proof: Let A be performed by processor i and B be performed by processor j. If $A \rightarrow B$, then the write to R_{ij} by A precedes the read of R_{ij} by B. Then the write of A to R_{ij} or a later write by action C of i to R_{ij} is a feasible write to the read of R_{ij} of B. Since R_{ij} is 0 self-stabilizing regular, this read should return the value of this write or a later write to R_{ij} by processor i during action C. Since processor i both reads and writes from R_{ii}, $\mathsf{label}(A) \leq \mathsf{max}(C) \leq \mathsf{label}(C)$. Therefore the read of R_{ij} by B returns a label greater than or equal to $\mathsf{label}(A)$. As B picks the maximum of all labels read, $\mathsf{label}(A) \leq \mathsf{max}(B)$. Then, if B is a read, $\mathsf{label}(A) \leq \mathsf{label}(B)$. If B is a write, then $\mathsf{label}(A) < \mathsf{label}(B)$. ■

Lemma 12 *For all $R \in \mathcal{R}$ there exists a $W \in \mathcal{W}$ such that $\mathsf{label}(W) = \mathsf{label}(R)$ and $R \not\rightarrow W$.*

Proof: In general if a read writes a label, it read that same label from a register. Let us write $A \rightsquigarrow B$ if the label read by B was written by A and $\mathsf{label}(A) = \mathsf{label}(B)$. If $A \rightsquigarrow B$, $B \rightsquigarrow C$ and B is a read, then define $A \rightsquigarrow C$. Now clearly $A \rightsquigarrow B$ implies $B \not\rightarrow A$ and $\mathsf{label}(A) = \mathsf{label}(B)$. Note that $\rightsquigarrow$ induces a *tree* of nodes A such that $A \rightsquigarrow B$.

Suppose there is no A such that $A \rightsquigarrow B$. Then $B \in \mathcal{F}$, because if some operation $C \rightarrow B$ on the same processor has $\mathsf{label}(C) = \mathsf{label}(B)$ then $C \rightsquigarrow B$, while if $\mathsf{label}(C) < \mathsf{label}(B)$ (the only other possible case according to lemma 11) then the contents of the register from which B obtains $\mathsf{label}(B)$ has changed after C read that same register. This register then must have been written by an operation D with $\mathsf{label}(D) = \mathsf{label}(B)$ before B reads it: $D \rightsquigarrow B$.

Now pick a $B \in \mathcal{A}$ such that $B \rightsquigarrow R$ and for no A, $A \rightsquigarrow B$. If no such B exists then set $B = R$. Then $B \in \mathcal{F}$, so $\mathsf{label}(B) \in \mathsf{label}(\mathcal{F})$. Hence either there exists a $W \in \mathcal{W}^-$ such that $\mathsf{label}(W) = \mathsf{label}(B) = \mathsf{label}(R)$, or $\mathsf{label}(B) \in \mathcal{F}_l$ so by $(F2)$ there exists a $W' \in \mathcal{F}_W$ with $\mathsf{label}(W') = \mathsf{label}(B) = \mathsf{label}(R)$. In the first case, by lemma 11, $R \not\rightarrow W$ as required. In the second case, since $B \in \mathcal{F}$ we must have by $(F4)$, $s(W') < s(B)$. Then as $B \rightsquigarrow R$ implies $R \not\rightarrow B$, this implies $R \not\rightarrow W'$. ■

Lemma 13 *For all $W, W' \in \mathcal{W}$ if $\mathsf{label}(W) = \mathsf{label}(W')$ then $W = W'$.*

Proof: There are three cases

$W, W' \in \mathcal{W}^-$**:** By the protocol then W and W' must be executed by the same processor, or else their *id*-fields differ, and thus $\mathsf{label}(W) \neq \mathsf{label}(W')$. But then either $W \to W'$ or $W' \to W$. By lemma 11, we must have $\mathsf{label}(W) < \mathsf{label}(W')$ or $\mathsf{label}(W') < \mathsf{label}(W)$. This is a contradiction.

$W \in W^-, W' \in \mathcal{F}_W$**:** If $W \in W^-$ then $\mathsf{label}(W) \notin \mathcal{F}_l$, while as $W' \in \mathcal{F}_W$ we must have $\mathsf{label}(W') \in \mathcal{F}_l$ by $(F1)$. Therefore $\mathsf{label}(W) \neq \mathsf{label}(W')$, a contradiction.

$W, W' \in \mathcal{F}_W$**:** If $\mathsf{label}(W) = \mathsf{label}(W')$, then by $(F3)$ we have $W = W'$.

■

Lemma 14 *If $A \to B$ and $B \in \mathcal{W}$ then $\mathsf{label}(A) < \mathsf{label}(B)$.*

Proof: If $B \in \mathcal{W}^-$ the result follows from lemma 11. So consider $B \in \mathcal{F}_W$. Note that the result follows from lemma 11 if we show $\mathsf{label}(A) \neq \mathsf{label}(B)$. There are three cases

$A \in \mathcal{F}$**:** the result follows from $(F4)$, or

$A \in \mathcal{W}^-$**:** then $\mathsf{label}(A) \notin \mathcal{F}_l$ while $\mathsf{label}(B) \in \mathcal{F}_l$ by $(F1)$ and the result follows, or

$A \in \mathcal{R}^-$**:** then from lemma 12 we conclude that there exists a $W \in \mathcal{W}$ with $\mathsf{label}(W) = \mathsf{label}(A)$ and $A \not\to W$. Then $W \neq B$ so from lemma 13 $\mathsf{label}(A) \neq \mathsf{label}(B)$.

■

Now we can define a reading mapping $\pi : \mathcal{R} \mapsto \mathcal{W}$ for a particular run $\langle \mathcal{A}, \to \rangle$ as follows. $\pi(R) = W$ if $\mathsf{label}(R) = \mathsf{label}(W)$ (and of course $W \in \mathcal{W}$). That this is a proper definition is shown in the next lemma

Lemma 15 *For all $R \in \mathcal{R}$, $\pi(R)$ is defined and unique, $R \not\to \pi(R)$, and R returns the value written by $\pi(R)$.*

Proof: That $\pi(R)$ is defined follows from lemma 12. That it is unique follows from lemma 13. That $R \not\to \pi(R)$ immediately follows from lemma 14.

If $\pi(R) \in \mathcal{W}^-$, then $\mathsf{label}(\pi(R)).val$ equals the value written by $\pi(R)$. If $\pi(R) \in \mathcal{F}_W$ we can arbitrarily define $\mathsf{label}(\pi(R)).val$ to be the value written by $\pi(R)$ (since for such $\pi(R)$ we must have $\mathsf{count}(\pi(R)) = 1$). ■

We now show that every run $\langle \mathcal{A}, \to \rangle$ with the above reading function π is atomic (i.e. linearizable). Define for $W \in \mathcal{W}$ its clan $[W]$ by $[W] = \{W\} \cup \{R \in \mathcal{R} \mid \pi(R) = W\}$, and let $\Gamma = \{[W] \mid W \in \mathcal{W}\}$ be the set of all clans. Define $\to'$ over Γ by

$$[W] \to' [W'] \iff (\exists A \in [W], B \in [W'] :: A \to B)$$

Lemma 16 *For all $W \in \mathcal{W}'$ and $A, B \in [W]$ we have* $\mathsf{label}(A) = \mathsf{label}(B)$. *Also if $W \neq W'$, then for all $A \in [W], B \in [W']$ we have* $\mathsf{label}(A) \neq \mathsf{label}(B)$.

Proof: The first part follows from the definition of $[W]$ and $\pi(R)$. The second part follows from lemma 13. ■

Lemma 17 *$\rightarrow'$ is an acyclic partial order over Γ.*

Proof: Suppose not. Then there exists a chain

$$[W_1] \rightarrow' [W_2] \rightarrow' \cdots \rightarrow' [W_m] \rightarrow' [W_1]$$

with $m > 1$, and $W_i \neq W_j$ if $i \neq j$. This implies that there exist actions $A_i, B_i \in [W_i]$ for all i with $1 \leq i \leq m$ such that $A_i \rightarrow B_{i+1}$ (addition modulo $m+1$ from now) for all i in $1 \leq i \leq m$. By lemma 11 $\mathsf{label}(A_i) \leq \mathsf{label}(B_{i+1})$ and by lemma 16 $\mathsf{label}(A_i) \leq \mathsf{label}(A_{i+1})$. Then $\mathsf{label}(A_1) = \mathsf{label}(A_2)$, contrary to lemma 16. ■

Applying these lemmas and the results of [AKKV88] we get the result.

Theorem 18 *Protocol 3 implements a 1-stabilizing $nWnR$ l-ary atomic register using n^2 0-stabilizing $1W1R$ ∞-ary regular registers.*

Proof: Define a total order $\Rightarrow$ over $\mathcal{A}$ extending $\rightarrow$ as follows. First extend $\rightarrow'$ over Γ to a total order $\Rightarrow'$; this is possible as $\rightarrow'$ is acyclic according to lemma 17. Now for $A \in [W]$ and $B \in [W']$ let $A \Rightarrow B$ if $[W] \Rightarrow' [W']$ (a). This extends $\rightarrow$ because if $A \rightarrow B$, then by the definition of $\rightarrow'$, $[W] \rightarrow' [W']$ and thus $[W] \Rightarrow' [W']$. For $A, B \in [W]$ fix an arbitrary extension $\Rightarrow$ of $\rightarrow$ such that for the only writer $W \in [W]$ we have for all other $C \in [W]$ that $W \Rightarrow C$ (b). This is an extension of $\rightarrow$ because by lemma 15, $C \not\rightarrow W$. Now $\Rightarrow$ is a total order over $\mathcal{A}$ such that for all $R \in \mathcal{R}$ $\pi(R) \Rightarrow R$ by lemma 15 and (b). Also there does not exists a $W \in \mathcal{W}$ such that $\pi(R) \Rightarrow W \Rightarrow R$, because by (a) and the fact that $R \notin [W]$ by lemma 16 either $W \Rightarrow [R]$ or $[R] \Rightarrow W$. Hence $W \Rightarrow \pi(R)$ or $R \Rightarrow W$. ■

7. Further research

There are still a lot of interesting questions in this new area that remain unanswered. First of all, our construction of the 1-stabilizing $nWnR$ atomic register uses unbounded timestamps to invalidate old values. We would like to know whether this necessarily so, or if the space requirements of a k-stabilizing atomic register can be bounded as in the non self-stabilizing case [IS92]. Second, following the work of Aspnes and Herlihy [AH90], it is an interesting venture to classify, based on their sequential specification, all k-stabilizing shared memory objects that can be constructed from k'-stabilizing atomic registers, and to provide a general method to do so.

Acknowledgements

It's a pleasure to thank Moti Yung; he encouraged this work.

REFERENCES

[AH90] ASPNES, J., AND HERLIHY, M. P. Wait-free data structures in the asynchronous PRAM model. In *2nd SPAA* (1990), pp. 340–349.

[AH93] ANAGNOSTOU, E., AND HADZILACOS, V. Tolerating transient and permanent failures. In *7th WDAG* (Lausanne, 1993), pp. 174–188.

[AKKV88] AWERBUCH, B., KIROUSIS, L. M., KRANAKIS, E., AND VITÁNYI, P. M. B. A proof technique for register atomicity. In *8th FST&TCS* (1988), pp. 286–303.

[AKM+93] AWERBUCH, B., KUTTEN, S., MANSOUR, Y., PATT-SHAMIR, B., AND VARGHESE, G. Time optimal self-stabilizing synchronization. In *25th STOC* (1993), pp. 652–661.

[AKY90] AFEK, Y., KUTTEN, S., AND YUNG, M. Memory-efficient self stabilizing protocols for general graphs. In *4th WDAG* (1990), pp. 15–28.

[BGW89] BROWN, G. M., GOUDA, M. G., AND WU, C. L. Token systems that self-stabilize. *IEEE Trans. on Comput.* **38**, 6 (1989), 845–852.

[BP89] BURNS, J. E., AND PACHL, J. Uniform self-stabilizing rings. *ACM Trans. Prog. Lang. & Syst.* **11**, 2 (1989), 330–344.

[Dij74] DIJKSTRA, E. W. Self-stabilizing systems in spite of distributed control. *Comm. ACM* **17**, 11 (1974), 643–644.

[DIM93] DOLEV, S., ISRAELI, A., AND MORAN, S. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distr. Comput.* **7**, 1 (1993), 3–16.

[DW93] DOLEV, S., AND WELCH, J. L. Wait-free clock synchronization. In *12th PODC* (1993), pp. 97–108.

[GP93] GOPAL, A. S., AND PERRY, K. J. Unifying self-stabilization and fault-tolerance. In *12th PODC* (Ithaca, 1993), pp. 195–206.

[Her91] HERLIHY, M. P. Wait-free synchronization. *ACM Trans. Prog. Lang. & Syst.* **13**, 1 (1991), 124–149.

[Hoe94] HOEPMAN, J.-H. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In *8th WDAG* (Terschelling, The Netherlands, 1994), pp. 265–279.

[IJ93] ISRAELI, A., AND JALFON, M. Uniform self-stabilizing ring orientation. *Inf. & Comput.* **104**, 2 (1993), 175–196.

[IS92] ISRAELI, A., AND SHAHAM, A. Optimal multi-writer multi-reader atomic register. In *11th PODC* (Vancouver BC, Canada, 1992), pp. 71–82.

[KK89] KIROUSIS, L. M., AND KRANAKIS, E. A brief survey of concurrent readers and writers. *CWI Quart.* **2**, 4 (1989), 307–330.

[KP90] KATZ, S., AND PERRY, K. J. Self-stabilizing extensions for message-passing systems. In *9th PODC* (1990), ACM, pp. 91–101.

[Lam86] LAMPORT, L. On interprocess communication. part I: basic formalism, part II: Algorithms. *Distr. Comput.* **1**, 2 (1986), 77–101.

[LTV89] LI, M., TROMP, J., AND VITÁNYI, P. M. B. How to share concurrent wait-free

variables. Tech. Rep. CS-R8916, CWI, 1989.

[LV92] LI, M., AND VITÁNYI, P. M. B. Optimality of wait-free atomic multiwriter variables. *Inf. Proc. Letters* **43** (1992), 107–112.

[PB87] PETERSON, G. L., AND BURNS, J. E. Concurrent reading while writing ii: The multi-writer case. In *28th FOCS* (1987), pp. 383–392.

[PT94] PAPATRIANTAFILOU, M., AND TSIGAS, P. Wait-free self-stabilizing clock synchronization. In *4th SWAT* (Århus, Denmark, 1994), pp. 267–277.

[Sch93] SCHNEIDER, M. Self-stabilization. *ACM Comput. Surv.* **25**, 1 (1993), 45–67.

[VA86] VITÁNYI, P. M. B., AND AWERBUCH, B. Atomic shared register access by asynchronous hardware. In *27th FOCS* (1986), pp. 233–243.