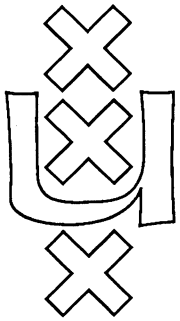
Institute for Language, Logic and Information

HOW TO SHARE CONCURRENT WAIT-FREE VARIABLES

Ming Li
John Tromp
Paul M.B. Vitanyi

ITLI Prepublication Series
for Computation and Complexity Theory CT-91-02



University of Amsterdam

The ITLI Prepublication Series

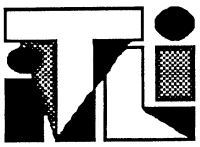
1986
86-01 The Institute of Language, Logic and Information
86-02 Peter van Emde Boas A Semantical Model for Integration and Modularization of Rules
86-03 Johan van Benthem Categorial Grammar and Lambda Calculus
86-04 Reinhard Muskens A Relational Formulation of the Theory of Types
86-05 Kenneth A. Bowen, Dick de Jongh Some Complete Logics for Branched Time, Part I Well-founded Time, Forward looking Operators
86-06 Johan van Benthem Logical Syntax

1987 87-01 Jeroen Groenendijk, Martin Stokhof Type shifting Rules and the Semantics of Interrogatives
87-02 Renate Bartsch Frame Representations and Discourse Representations
87-03 Jan Willem Klop, Roel de Vrijer Unique Normal Forms for Lambda Calculus with Surjective Pairing
87-04 Johan van Benthem Polyadic quantifiers
87-05 Víctor Sánchez Valencia Traditional Logicians and de Morgan's Example
87-06 Eleonore Oversteegen Temporal Adverbials in the Two Track Theory of Time
87-07 Johan van Benthem Categorial Grammar and Type Theory
87-08 Renate Bartsch The Construction of Properties under Perspectives
87-09 Herman Hendriks Type Change in Semantics: The Scope of Quantification and Coordination

1988 LP-88-01 Michiel van Lambalgen *Logic, Semantics and Philosophy of Language:* Algorithmic Information Theory
LP-88-02 Yde Venema Expressiveness and Completeness of an Interval Tense Logic
LP-88-03 Year Report 1987
LP-88-04 Reinhard Muskens Going partial in Montague Grammar
LP-88-05 Johan van Benthem Logical Constants across Varying Types
LP-88-06 Johan van Benthem Semantic Parallels in Natural Language and Computation
LP-88-07 Renate Bartsch Tenses, Aspects, and their Scopes in Discourse
LP-88-08 Jeroen Groenendijk, Martin Stokhof Context and Information in Dynamic Semantics
LP-88-09 Theo M.V. Janssen A mathematical model for the CAT framework of Eurotra
LP-88-10 Anneke Kleppe A Blissymbolics Translation Program
ML-88-01 Jaap van Oosten *Mathematical Logic and Foundations:* Lifschitz' Realizabiility
ML-88-02 M.D.G. Swaen The Arithmetical Fragment of Martin Löf's Type Theories with weak Σ-elimination
ML-88-03 Dick de Jongh, Frank Veltman Provability Logics for Relative Interpretability
ML-88-04 A.S. Troelstra On the Early History of Intuitionistic Logic
ML-88-05 A.S. Troelstra Remarks on Intuitionism and the Philosophy of Mathematics
CT-88-01 Ming Li, Paul M.B.Vitanyi *Computation and Complexity Theory:* Two Decades of Applied Kolmogorov Complexity
CT-88-02 Michiel H.M. Smid General Lower Bounds for the Partitioning of Range Trees
CT-88-03 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas Maintaining Multiple Representations of Dynamic Data Structures
CT-88-04 Dick de Jongh, Lex Hendriks Gerard R. Renardel de Lavalette Computations in Fragments of Intuitionistic Propositional Logic
CT-88-05 Peter van Emde Boas Machine Models and Simulations (revised version)
CT-88-06 Michiel H.M. Smid A Data Structure for the Union-find Problem having good Single-Operation Complexity
CT-88-07 Johan van Benthem Time, Logic and Computation
CT-88-08 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas Multiple Representations of Dynamic Data Structures
CT-88-09 Theo M.V. Janssen Towards a Universal Parsing Algorithm for Functional Grammar
CT-88-10 Edith Spaan, Leen Torenvliet, Peter van Emde Boas Nondeterminism, Fairness and a Fundamental Analogy
CT-88-11 Sieger van Denneheuvel, Peter van Emde Boas Towards implementing RL
X-88-01 Marc Jumelet *Other prepublications:* On Solovay's Completeness Theorem

1989 LP-89-01 Johan van Benthem *Logic, Semantics and Philosophy of Language:* The Fine-Structure of Categorial Semantics
LP-89-02 Jeroen Groenendijk, Martin Stokhof Dynamic Predicate Logic, towards a compositional, non-representational semantics of discourse
LP-89-03 Yde Venema Two-dimensional Modal Logics for Relation Algebras and Temporal Logic of Intervals
LP-89-04 Johan van Benthem Language in Action
LP-89-05 Johan van Benthem Modal Logic as a Theory of Information
LP-89-06 Andreja Prijatelj Intensional Lambek Calculi: Theory and Application
LP-89-07 Heinrich Wansing The Adequacy Problem for Sequential Propositional Logic
LP-89-08 Víctor Sánchez Valencia Peirce's Propositional Logic: From Algebra to Graphs
LP-89-09 Zhisheng Huang Dependency of Belief in Distributed Systems
ML-89-01 Dick de Jongh, Albert Visser *Mathematical Logic and Foundations:* Explicit Fixed Points for Interpretability Logic
ML-89-02 Roel de Vrijer Extending the Lambda Calculus with Surjective Pairing is conservative
ML-89-03 Dick de Jongh, Franco Montagna Rosser Orderings and Free Variables
ML-89-04 Dick de Jongh, Marc Jumelet, Franco Montagna On the Proof of Solovay's Theorem
ML-89-05 Rineke Verbrugge Σ-completeness and Bounded Arithmetic
ML-89-06 Michiel van Lambalgen The Axiomatization of Randomness
ML-89-07 Dirk Roorda Elementary Inductive Definitions in HA: from Strictly Positive towards Monotone
ML-89-08 Dirk Roorda Investigations into Classical Linear Logic
ML-89-09 Alessandra Carbone Provable Fixed points in $I\Delta_0+\Omega_1$
CT-89-01 Michiel H.M. Smid *Computation and Complexity Theory:* Dynamic Deferred Data Structures
CT-89-02 Peter van Emde Boas Machine Models and Simulations
CT-89-03 Ming Li, Herman Neuféglise, Leen Torenvliet, Peter van Emde Boas On Space Efficient Simulations
CT-89-04 Harry Buhrman, Leen Torenvliet A Comparison of Reductions on Nondeterministic Space
CT-89-05 Pieter H. Hartel, Michiel H.M. Smid Leen Torenvliet, Willem G. Vree A Parallel Functional Implementation of Range Queries
CT-89-06 H.W. Lenstra, Jr. Finding Isomorphisms between Finite Fields
CT-89-07 Ming Li, Paul M.B. Vitanyi A Theory of Learning Simple Concepts under Simple Distributions and Average Case Complexity for the Universal Distribution (Prel. Version)
CT-89-08 Harry Buhrman, Steven Homer Leen Torenvliet Honest Reductions, Completeness and Nondeterminstic Complexity Classes
CT-89-09 Harry Buhrman, Edith Spaan, Leen Torenvliet On Adaptive Resource Bounded Computations
CT-89-10 Sieger van Denneheuvel The Rule Language RL/1
CT-89-11 Zhisheng Huang, Sieger van Denneheuvel Peter van Emde Boas Towards Functional Classification of Recursive Query Processing
X-89-01 Marianne Kalsbeek *Other Prepublications:* An Orey Sentence for Predicative Arithmetic
X-89-02 G. Wagemakers New Foundations: a Survey of Quine's Set Theory
X-89-03 A.S. Troelstra Index of the Heyting Nachlass
X-89-04 Jeroen Groenendijk, Martin Stokhof Dynamic Montague Grammar, a first sketch
X-89-05 Maarten de Rijke The Modal Theory of Inequality
X-89-06 Peter van Emde Boas Een Relationele Semantiek voor Conceptueel Modelleren: Het RL-project

1990 SEE INSIDE BACK COVER



Instituut voor Taal, Logica en Informatie
Institute for Language, Logic and Information

Faculteit der Wiskunde en Informatica
(Department of Mathematics and Computer Science)
Plantage Muidergracht 24
1018TV Amsterdam

Faculteit der Wijsbegeerte
(Department of Philosophy)
Nieuwe Doelenstraat 15
1012CP Amsterdam

HOW TO SHARE CONCURRENT WAIT-FREE VARIABLES

Ming Li
Computer Science Department
University of Waterloo

John Tromp
Paul M.B. Vitanyi
Department of Mathematics and Computer Science
University of Amsterdam
& CWI

ITLI Prepublication Series
for Computation and Complexity Theory
ISSN 0924-8374

Received March 1991

How to Share Concurrent Wait-Free Variables

Ming Li*
Computer Science Department, University of Waterloo
Waterloo, Ontario, Canada.

John Tromp
Paul M.B. Vitanyi
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Faculteit Wiskunde en Informatica
Universiteit van Amsterdam

February 27, 1991

Abstract

Sharing data between multiple asynchronous users—each of which can both read and write the data—such that the accesses are serializable and free from waiting, is a feature which may help to increase the amount of parallellism in distributed systems. An algorithm implementing this feature is presented. Using a structured, top-down approach, we obtain a better understanding of what the algorithms do, why they do it, and that they correctly implement the specification. The main construction of an n-user atomic variable directly from single-writer, single-reader atomic variables uses $O(n)$ control bits per subvariable and $O(n)$ accesses to subvariables per read/write action.

1980 Mathematics Subject Classification: 68C05, 68C25, 68A05, 68B20.
CR Categories: B.3.2, B.4.3, D.4.1, D.4.4.
Keywords and Phrases: Shared variable (register),
concurrent reading and writing, atomicity, multiwriter variable, simulation.

---

*M. Li was supported in part by the National Science Foundation under Grant DCR-8606366 at Ohio State University, by Office of Naval Research Grant N00014-85-K-0445 and Army Research Office Grant DAAL03-86-K-0171 at Harvard University, and by NSERC Grant OGP-0036747 at York University. The basic algorithms and proofs presented here were first announced in [9] and presented at the *International Conference on Automata, Languages, and Programming*, Stresa, Italy, 1989. E-mail adresses: mli@water.waterloo.edu, tromp@cwi.nl and paulv@cwi.nl

1 Introduction

In [8] Lamport has shown how an atomic variable—one whose accesses appear to be indivisible—shared between one writer and one reader, acting asynchronously and without waiting, can be constructed from lower level hardware rather than just assuming its existence. There arises the question of the construction of multi-user atomic variables of that type [17]. In this paper we will supply a uniform solution to such problems, given Lamport's construction, and derive the implementations by transformations from the specification. Independently from this derivation, we give a direct proof of the main result.

1.1 Informal Problem Statement and Main Result

Usually, with asynchronous readers and writers, atomicity of operations is simply assumed or enforced by synchronization primitives like semaphores. However, active serialization of asynchronous concurrent actions always implies waiting by one action for another. In contrast, our aim is to realize the maximum amount of parallellism inherent in concurrent systems by avoiding waiting altogether in our algorithms. In such a setting, serializability is *not* actively enforced, rather it is the result of a pre-established harmony in the way the executions of the algorithm by the various processors interact. Any one of the references, say [8] or [17], describes the problem area in some detail.

The point of departure is the solution of the following problem. (We keep the discussion informal). A flip-flop is a Boolean variable that can be read (tested) by one processor and written (set, reset, or changed) by another. Suppose, one is given atomic flip-flops as building blocks, and is asked to implement an atomic variable with range 0 to $n-1$, that can be written by one processor and read by another one. Of course, $\lceil \log_2 n \rceil$ flip-flops suffice to hold such a value. It is stipulated that the two processors are asynchronous and do not wait for one another. Suppose the writer gets stuck after it has written half the bits of the new value. If the reader executes a read while the writer is stuck, it obtains a value that consists of half the new value and half the old one. Obviously, this violates atomicity.

Such atomic variables, correctly implemented [13, 8], serve as the building blocks for our main construction. The following naming convention is used for classifying the accessibility of a variable (also called register):

single-user a local variable

single-reader one writer and another reader

multi-reader one writer and multiple readers

multi-writer multiple writers and readers

multi-user each user can both write and read

This constitutes a hierarchy of variables[1]. At the outset we state our main result:

Theorem 1 *An atomic wait-free n-user variable is constructed from $O(n^2)$ atomic wait-free 1-reader 1-writer variables, using $O(n)$ accesses of subvariables per read/write and $O(n)$ control bits per subvariable.*

1.2 Comparison with Related Work.

Related constructions are given by [15, 7, 3, 11, 6] for the single-reader to multi-reader case (which is dealt with in appendix B), and by [17, 12, 14] for the multi-reader to multi-writer case (see appendix A). The latter problem is the more difficult one. The solutions in references [17, 12] are known to be incorrect [14]. There has been no previous attempt to implement an n-user variable directly from single reader variables. Yet we believe the construction presented is relatively simple and transparent. Both problems above are solved by simplifications (as it were "projections") of our main solution. The precise form of our solution was arrived at by implementing and empirically testing several candidates. Appendix C discusses the *simulation* of our main algorithms.

The algorithm uses $O(n)$ accesses to single-reader subvariables per operation, and each single-reader subvariable stores two copies of the value of the constructed variable together with $O(n)$ bits of control information. In comparison, the multi-writer algorithm in [14] uses $\Omega(n^2)$ accesses to multi-reader variables per operation. Following [6], recent work [4] aims at providing a general method for replacing unbounded timestamps by bounded timestamps in concurrent systems of multi-reader variables. Application to a multi-writer variable requires $\Omega(n^2 \log n)$ accesses to multi-reader variables per operation. Other related work is [1, 2, 5, 8, 10, 13, 16].

1.3 Definitions

A concurrent system consists of a collection of sequential processes that communicate through shared data structures. The most basic such data structure is a shared variable. A user of such a variable V can start an action a (read or write) at any time when it is not engaged in another action, by invoking an "execute a" command on V, which finishes at some later time, possibly returning the value read. The semantics can be expressed in terms of a local value v of a process P and the global value contained in V. In absence of any other concurrent action the result of process P writing its local value v to V is that $V{:=}v$ is executed, and the result of a process reading the global V is that $v{:=}V$

[1] Although a multi-user variable can be trivially implemented by a multi-writer variable (and is not more powerful in that sense), this requires in the worst case a quadrupling in the number of subvariables ("space") used. It also seems impractical to make a distinction between users restricted to write actions and users restricted to read actions.

is executed. To emphasize the distinction between actions at a higher level, and those at a lower level, the word *operation execution*, or shortly *action*, is used for the former, and *subaction* is used for the latter.

An *implementation* of V consists of a set of *protocols*, one for each user process, and a set of single-reader subvariables $X, Y, \ldots, Z$. An operation execution a by user process P on V consists of an execution of the associated protocol in which it applies some transformations on the subvariables $X, Y, \ldots, Z$, followed by returning a result to P. An implementation is *wait-free* if the number of subvariable accesses in an operation execution is bounded by a constant, which depends only on the number of users.

Linearizability or atomicity is defined in terms of equivalence with a sequential system in which actions are mediated by a sequential scheduler that permits only one operation at a time to execute on any variable. A shared variable is *atomic*, if each read and write of it actually happens, or appears to take effect, instantaneously at some point between its invocation and response, irrespective of its actual duration. This can be formalized as follows.

Let V be a shared variable with associated user processes P,Q,...,R. Let $\mathcal{A}$ be the set of operation executions that the users execute on V. Each execution is comprised of a number of subvariable accesses, called subactions. Let $\rightarrow$ be the (total) order in which these subactions occur. Extend $\rightarrow$ to a partial order on sets of subactions, as follows:

Definition 1 Let $\rightarrow$ be a total order on all the subactions. Let S be a finite, nonempty set of subactions. Denote the $\rightarrow$-first subaction in S by $s(S)$ (start of S), and the $\rightarrow$-last subaction in S by $f(S)$ (finish of S).

Definition 2 Let $\rightarrow$ be a total order on all the subactions. Let S_1, S_2 be two finite, nonempty sets of subactions. Define: $S_1 \rightarrow S_2$ iff $f(S_1) \rightarrow s(S_2)$.

For convenience, a single subaction is interpreted as a singleton set, of which it is both the start and finish. Note that with this definition, $\rightarrow$ is a special type of partial order called an *interval* order. That is, a transitive binary relation such that if $a \rightarrow b$ and $c \rightarrow d$ then $a \rightarrow d$ or $c \rightarrow b$ (the *interval axiom*).

In the remainder of this section, consider as sets of subactions only those corresponding to the operation executions in $\mathcal{A}$.

Given the finite domain of a variable, sufficiently long runs of operations will have to contain multiple writes of the same value. But since the actual values written to the shared variable are not used, apart from being copied [2], a unique identity can be associated with each written value. In this way, for any action a, $val(a)$ is taken to be the uniquely identified value written/read by a. Define the reading mapping π as a mapping from reads to writes by: if r is a read that returns the value written by write w ($val(r) = val(w)$), then $\pi(r) = w$. The triple $\sigma = (\mathcal{A}, \rightarrow, \pi)$ is called a *system execution.*

[2] Peterson and Burns use the word *opaque* for this property.

Definition 3 A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ is *atomic* if $\rightarrow$ can be extended to a total order $\Rightarrow$ such that

(A1) $\pi(r) \Rightarrow r$, and

(A2) there is no write w such that $\pi(r) \Rightarrow w \Rightarrow r$.

That is, the partially ordered set of actions can be linearized while respecting the logical read/write order. A shared variable is *atomic* if each system execution of it is atomic.

1.4 The Problem to be Solved

The goal is to implement an atomic wait-free shared variable V for n users $0, \ldots, n-1$, such that each user can perform both reads and writes. V is implemented using atomic variables $R_{i,j} (0 \leq i, j < n)$, for which user i is the only associated writer process and user j is the only associated reader process. Since only user i can write to variables $R_{i,0}, \ldots, R_{i,n-1}$, it *owns* these variables.

1.5 Specification

While the definition of atomicity is quite clear, it is convenient to transform it into an equivalent specification, from which the first algorithm that implements V can be directly derived. Call two actions a and b equivalent, $a \equiv b$, if $val(a) = val(b)$. Note that for a write action w, its equivalence class is $[w] = \{w\} \cup \pi^{-1}(w)$, consisting of exactly one write (itself), and 0 or more reads which obtain the value it writes. The precedence relation $\rightarrow$ on $\mathcal{A}$ induces a relation $\ll$ on $\mathcal{A}/\equiv$ (the set of equivalence classes) as follows:

Definition 4 For two writes $w_1, w_2 \in \mathcal{A}$, $[w_1] \ll [w_2]$ iff $w_1 \neq w_2$ and there exist $a_1 \equiv w_1$ and $a_2 \equiv w_2$ such that $a_1 \rightarrow a_2$.

The following lemma comes from [1]:

Lemma 1 *Let $(\mathcal{A}, \rightarrow, \pi)$ be a system execution. There exists an extension $\Rightarrow$ of $\rightarrow$ satisfying (A1) and (A2) iff $\ll$ is acyclic and not $r \rightarrow \pi(r)$ for any read r.*

Proof. "If". If $\ll$ has no cycles then it can be extended to a total order $<$ on $\mathcal{A}/\equiv$. Define $\Rightarrow$ on $\mathcal{A}$ by:

1. if $a \not\equiv b$ then $a \Rightarrow b$ iff $[a] < [b]$
2. within each $[w]$, topologically sort the elements beginning with the write (using $r \not\rightarrow \pi(r)$), so that $[w] = \{w, r_1, \ldots, r_k\}$ and $r_i \rightarrow r_j$ implies $i < j$. Now put $w \Rightarrow r_1 \Rightarrow r_2 \Rightarrow \cdots \Rightarrow r_k$.

We claim that $\Rightarrow$ is an extension of $\rightarrow$. Assume that $a \rightarrow b$. Then either $a \not\equiv b$ and by definition of $\ll$ it follows that $[a] \ll [b]$ hence $[a] < [b]$ and thus $a \Rightarrow b$, or $a \equiv b$ in which case $a \Rightarrow b$ follows from the topological sort. Furthermore, $\Rightarrow$ is a total order on $\mathcal{A}$ since the elements in each $[w]$ are totally ordered and the $[w]$'s themselves are totally ordered by 1. Finally, (A1) and (A2) hold because $\pi(r)$ is the last write $\Rightarrow$-preceding r.

"Only if". First assume $r \rightarrow \pi(r)$ for some read r. Since $\Rightarrow$ extends $\rightarrow$, also $r \Rightarrow \pi(r)$, which is in contradiction with (A1). Therefore, $r \not\rightarrow \pi(r)$ for any read r. Because of (A1) and (A2) each $[w]$ is a consecutive sequence of actions in the total order $\Rightarrow$. It follows that the order $<$ on the $[w]$'s induced by $\Rightarrow$ is total. If $[w_1] \ll [w_2]$ then there are $a_1 \equiv w_1$ and $a_2 \equiv w_2$ such that $a_1 \rightarrow a_2$. Since $\Rightarrow$ extends $\rightarrow$, also $a_1 \Rightarrow a_2$ and thus $[w_1] < [w_2]$. Then, $<$ is an extension of $\ll$. Since $<$ is acyclic, so must $\ll$ be. □

This gives the specification that an atomic variable has to satisfy: for each of its system executions $\sigma = (\mathcal{A}, \rightarrow, \pi)$

(S1) not $r \rightarrow \pi(r)$ for any read action r, and

(S2) the induced relation $\ll$ has no cycles.

2 The Basic Algorithm

The first approximation of the target algorithm captures the essence of the problem solution apart from the boundedness of the constituent shared variables (see also [17]). Let V be as in the problem description above. Violation of (S1) would mean that a read execution returns a value before the write of it ever started. This condition will be trivially satisfied by all algorithms considered here. How can (S2) be satisfied? Proceed as follows. Let $(T, <)$ be a totally ordered set of *tags*. For each system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$, let $tag : \mathcal{A} \rightarrow T$ be a function such that

(T1) if $a \equiv b$ then $tag(a) = tag(b)$

(T2) if $a \rightarrow b$ then $tag(a) \leq tag(b)$, and

(T3) if $w_1 \neq w_2$ are both writes, then $tag(w_1) \neq tag(w_2)$.

(T1) ensures a single tag for equivalent actions, so that we may write $tag([a])$. If $[w_1] \ll [w_2]$ then by definition $w_1 \neq w_2$ and there exist $a \equiv w_1$ and $b \equiv w_2$ with $a \rightarrow b$. Apply (T3) and (T2) respectively, to obtain $tag([w_1]) \neq tag([w_2])$ and $tag([w_1]) \leq tag([w_2])$, hence $tag([w_1]) < tag([w_2])$. Therefore, $\ll$ has no cycles and (S2) is satisfied. Hence we have shown:

Lemma 2 *Let V be a variable such that for each system execution σ condition (S1) is satisfied, and there is a function tag satisfying (T1), (T2), and (T3). Then V is atomic.*

Using $T = \mathbb{N}$, the set of nonnegative integers, an atomic variable V can be implemented as follows. To satisfy

T1: read-actions copy the tag of the return-value

T2: actions maximize over the visible tags

T3: write actions choose a tag

- greater than the maximum visible tag
- congruent to the user modulo n

The last two items ensure different tags for the writes of a single user, and those of different users (recall that the users are numbered $0, \ldots, n-1$). To make explicit the congruence modulo n one can view a tag t as a pair (ts, id), where $t = n \cdot ts + id$ and $0 \leq id < n$. Call $ts = t$ div n the *timestamp* and $id = t$ mod n the *index*.

Figure 1 shows the basic algorithm. The architecture is an n by n matrix of atomic subvariables $R_{i,j}$, providing a directed communication path from user i to user j. The subvariable $R_{i,j}$ can be written by user i in a statement `Write` $R_{i,j}$ `:= loc`, where `loc` is a local variable. Likewise, it can be read by user j in a statement `Read loc :=` $R_{i,j}$. Each $R_{i,j}$ contains the fields `value` and `tag`.

This and all other algorithms in this paper are initialized by simply setting all fields of all local and subvariables to 0. This puts the system in a state which appears to have resulted from an initial write by user 0 of the value 0 with a zero tag, followed by successive reads of the other users $1, \ldots, n-1$, all of which choose `max := 0`.

Locally, each user i has an array `from[0],...,from[n-1]` where `from[j]` is to hold a local copy of $R_{j,i}$. Note that a single protocol is given for both read and write actions. All multi-user algorithms presented in this paper have such a unified protocol.

A `return` statement exits the execution of the protocol. The argument of the `return` statement is used by read actions as the return value, and is ignored by write actions. If the end of a protocol can only be reached by a write action, then a `return` statement in the last line is superfluous and omitted.

Lemma 3 *Algorithm 0 implements an atomic wait-free multi-user variable.*

Proof. (See also [1, 17].) Obviously Algorithm 0 is wait-free. One only has to argue atomicity. Let $\sigma = (\mathcal{A}, \rightarrow, \pi)$ be a system execution according to Algorithm 0. (T1) is satisfied since tags are copied alongside their corresponding values.

Claim 1 For all i, j $(0 \leq i, j < n)$, $R_{i,j}$.tag is nondecreasing in time.

```
 1. for j:=0 to n-1 do Read from[j] := R_{j,i}
 2. select max such that ∀j (from[j].tag ≤ from[max].tag)
 3. if read_action then
 4.   from[i].value := from[max].value
 5.   from[i].tag := from[max].tag
 6. else if write_action then
 7.   from[i].value := newvalue
 8.   from[i].tag := n * (from[max].tag div n + 1) + i
 9. endif
10. for j:=0 to n-1 do Write R_{i,j} := from[i]
11. return from[i].value
```

Figure 1: Algorithm 0; protocol for user i

Proof. Consider first `from[i].tag` of user i. It is only changed in lines 5 and 8. The new value of `from[i].tag` is at least `from[max].tag`, which by line 2 is at least the old value of `from[i].tag`. Since $R_{i,j}$.tag only changes when user i overwrites it with `from[i].tag` (line 10), the claim holds for these fields as well. □

Moreover, in a write action, the new timestamp of i is at least 1 greater than the previous timestamp, and the new index equals i, so two different write actions, either by the same user or by two different users, must have different tags, thus satisfying (T3). Let $a \rightarrow b$ be two actions by users i and j respectively. Due to the maximizing in line 2, the new tag of action b is at least `from[i].tag` which, according to claim 1, is at least the tag of a, since a writes its tag to $R_{i,j}$ and finishes before b starts. Therefore, (T2) holds as well. Condition (S1) is satisfied trivially. By lemma 2, Algorithm 0 is atomic. □

3 Solution Method

The only problem with Algorithm 0 is that $T = I\!N$ is infinite. Through a series of transformations of the basic algorithm, this shortcoming is removed.

In order to motivate the first transformation, consider the following system execution scenario.

User 0 starts a write operation, but falls asleep after writing to $R_{0,1}$ in line 10. Then user 1 also starts a write operation, sees the new tag $(1,0)$ of user 0, and falls asleep after writing to $R_{1,2}$ in line 10. This continues in the obvious manner until user $n-3$ writes to $R_{n-3,n-2}$ and also falls asleep. Clearly, none of the writing users have informed user $n-1$ of the ongoing activities, and as a result the maximum timestamp in column $n-2$ $(R_{0,n-2}, \ldots, R_{n-1,n-2})$ is approximately n greater than the maximum timestamp in column $n-1$. It will later prove useful to bound these differences. Algorithm 1 (figure 2) takes care of this by having the write actions also "propagate" the value and tag that

they consider to be most recent. In this way, the write protocol becomes an extension of the read protocol.

```
1. for j:=0 to n-1 do Read from[j] := R_{j,i}
2. select max such that ∀j (from[j].tag ≤ from[max].tag)
3. from[i].value := from[max].value
4. from[i].tag := from[max].tag
5. for j:=0 to n-1 do Write R_{i,j} := from[i]
6. if read_action then return from[i].value
7. from[i].value := newvalue
8. from[i].tag := n * (from[max].tag div n + 1) + i
9. for j:=0 to n-1 do Write R_{i,j} := from[i]
```

Figure 2: Algorithm 1; protocol for user i

The correctness of Algorithm 1 follows easily from that of Algorithm 0. We do not need a proof of this for the sequel, since we prove the correctness of the next Algorithm 2 from scratch.

3.1 Outline of the Algorithm

The obvious way to proceed is to refine Algorithm 1 to an algorithm that induces a function tag' such that for each action a one has $tag'(a) \equiv tag(a) \pmod{C}$, for some system constant C. The *only* difficulty with this scheme is that the old tags may be confused with the new ones when looping back. However, if the loop is wide enough, then outdated tags are scanned many times by at least one user. Similarly, long pending actions are scanned many times by at least one user. Intuitively, this can be exploited as follows.

Each time a user finishes a write, it 'shoots' every other read or write execution it 'sees' once. An action is discredited, if it is shot at least d times by the same user, d a large enough system constant. In this way, each out of date action gets discredited after at most about dn writes. This solves the problem of discrediting actions which are already outdated when they are scanned.

One must also solve the problem of actions which are not outdated when scanned initially, but become outdated in the course of the scanning phase, simply because it takes too long. This can be detected and resolved by having the scanning action check the number of times it has been shot by each user, subsequent to the scanning phase.

To implement all this, add new lines to Algorithm 1. The old line 1 of Algorithm 1 is 'bracketed' by an extra preliminary phase that sets up a target associated with the operation execution, and by an extra test phase that checks the number of times the target has been shot by whom. This way an operation can check whether or not it has been shot at least e times by the same user. If it has, then it completely 'overlaps' a write execution from this set, say the one-

before-last, and can safely 'abort', terminate without executing the remainder of its protocol. Hence, at most about en writes complete during the scanning phase of a non-aborting operation execution a. Taking into account the 'about' caveats in the discussion, a loop of length C, using, say C div $n = 2(d+e+1)n+2$ timestamps, suffices to identify the largest element of the set of credible tags, clustered in a $n((d+e+1)n+1)$-size 'window'.

An aborting read execution will report the value of the one-before-last shooting write of a user who causes the abortion. An aborting write action will simply not write at all. Intuitively, an aborting read can be ordered just after the write whose value it reports. This requires that the value written by the one-before-last write is available. It is therefore saved in the preliminary phase added to Algorithm 1. The shooting phase is inserted just before the end of Algorithm 1. An aborting write can be ordered just before the one-before-last shooting write of a user who causes the abortion.

3.2 Outline of the Proof

Inserting the above additions in Algorithm 1, we obtain Algorithm 2. Actions do not yet get discredited, but they do abort (with $e = 3$). Just like in the proof of Algorithm 0, we show (from scratch) that for each system execution σ of Algorithm 2 there is a function *tag* satisfying (T1), (T2), and (T3). The change between Algorithm 2 and the next Algorithm 3 consists in discrediting actions (with $d = 5$). It is shown that the change from Algorithm 2 to Algorithm 3 leaves the associated function *tag* invariant. The next step is to show that in Algorithm 3 all comparisons between unbounded variables (like timestamps) are between comparands which differ by at most a function of the number of users (like $9n + 1$). Simply reducing all such variables modulo the appropriate number yields the final Algorithm 4 using only bounded variables, and, again, leaves the associated function *tag* invariant. Condition (S1) holds trivially. By lemma 2, it is established that Algorithm 4 is atomic. It is wait-free, because the protocol does not contain unbounded loops. This establishes theorem 1, but for the complexity aspects which are easy to verify.

4 Algorithm 2

Looking at figure 3, it is observed that the implementation of the shooting feature adds some complexity to the algorithm. All the old lines of Algorithm 1 are still present as indicated, with line 2 (Algorithm 1) rephrased as lines 12,13 (Algorithm 2), making the new algorithm an extension of Algorithm 1. In the added lines, which we have put in boxes, one can see some new fields in the registers. The field previous is used to hold the value of the last completed write (line 1). The total number of shots from user j to user i's action number $a(= 0, 1, 2, \ldots)$ is counted on shoot$[i][a]$ of user j. The total amount of healing

```
1.  from[i].previous := from[i].value
2.  s := 1 - from[i].ss
3.  for j:=0 to n-1 do
4.    Read tmp := R_{j,i}                                          e(xamine)
5.    from[i].heal[j][s] := tmp.shoot[i][s]
6.  { from[i].num +:= 1 }
7.  for j:=0 to n-1 do Write R_{i,j} := from[i]                    h(eal)
8.  for j:=0 to n-1 do Read from[j] := R_{j,i}                     r(ead)
9.  for j:=0 to n-1 do
10.   Read tmp := R_{j,i}                                          t(est)
11.   if tmp.shoot[i][s] - from[i].heal[j][s] ≥ 3 then return tmp.previous
12. L := { 0,...,n-1 }
13. select max ∈ L such that ∀j ∈ L (from[j].tag ≤ from[max].tag)
14. from[i].value := from[max].value
15. from[i].tag := from[max].tag
16. from[i].ss := s
17. { from[i].pnum := from[i].num }
18. for j:=0 to n-1 do Write R_{i,j} := from[i]                    p(ropagate)
19. if read_action then return from[i].value
20. from[i].value := newvalue
21. from[i].tag := n * (from[max].tag div n + 1) + i
22. for j:=0 to n-1 and for s:=0 to 1 do
23.   if from[i].shoot[j][s] - from[j].heal[i][s] < 5
24.   then from[i].shoot[j][s] +:= 1
25. for j:=0 to n-1 do Write R_{i,j} := from[i]                    w(rite)
```

Figure 3: Algorithm 2; protocol for user i

of user i's action number a with respect to user j is counted on `heal[`j`][`a`]` of user i. The difference between these two counters, `shoot[`i`][`a`]` $-$ `heal[`j`][`a`]`, at any time represents the number of shots that user i's action a has received from user j. At any state of the system execution, there are only two relevant actions a, namely, the one in progress and the previous one. Thus, it suffices to use `shoot[`i`][`s`]` and `heal[`j`][`s`]`, with $s = a \bmod 2$. The value of s for a non-aborting action is recorded in the 'shoot selector' field `ss`. The values in the shoot and heal counters, like the tags, are unbounded. It will be seen later that they can be easily bounded. However, doing so at this point would only complicate the algorithm and its discussion. In detail, the shooting works as follows. At the start of a new action, user i complements its local variable s, and reads the `shoot[i][s]` entry of $R_{j,i}$. Then it sets its `heal[j][s]` equal to that shoot count, so as to heal from shots delivered to its previous actions. During the reading phase (line 8) of i's action, user j may perform some write actions. At the end of each such action, user j checks whether the difference between `shoot[i][s]` and `heal[j][s]` has reached 5 yet. If not then it shoots the current action of user i by incrementing (`+:= 1`) its shoot counter in line 24. In this way the difference between a shoot counter and the corresponding heal counter is always between 0 and 5 inclusive.

Consider the number of shots to *wound* an action (3). After the reading phase, user i reads the `shoot[i][s]` entry of $R_{j,i}$ again and tests wether the difference with its `heal[j][s]` is at least 3. If so, then the action aborts, i.e. it exits execution of the protocol, returning the value of a recently completed write action by user j in line 11. (Recall our convention that execution of a return statement terminates the operation execution, and its argument is ignored if the operation is a write.) Otherwise, the normal course of action is resumed.

The fields num and pnum serve no other purpose than simplifying the proofs. They number the actions of each user from 0 onwards. The auxiliary lines 6 and 17 are placed between braces {,} to emphasize that they are not part of the algorithm itself.

4.1 Proof Preliminaries

The set of subactions comprising an execution of Algorithm 2 can be divided into 6 *phases*, each consisting of n subreads/subwrites. These are indicated in figure 3 at the right end of those lines which involve shared register access. Denote the phases by a single letter extension to the action name, and the subactions in each phase by a subscript. E.g., if a is an action by user 3, then $a.\mathrm{t}_2$ is the subread of $R_{2,3}$ in line 10; the testing phase. A complete execution of the algorithm, say action a by user i, consists of the phases

$$a.\mathrm{e} \rightarrow a.\mathrm{h} \rightarrow a.\mathrm{r} \rightarrow a.\mathrm{t} \rightarrow a.\mathrm{p} \rightarrow a.\mathrm{w},$$

where each phase $a.c$ is a sequence of subactions

$$a.c_0 \rightarrow \cdots \rightarrow a.c_{n-1}.$$

Recall the definition of start $s()$ and finish $f()$ (definition 1). Using the notation above, we have $s(a) = a.e_0$. Moreover, $f(a) = a.t_j$ for an action aborted by user j, $f(a) = a.p_{n-1}$ for a non-aborting read, and $f(a) = a.w_{n-1}$ for a non-aborting write.

We first prove an extension of claim 1.

Claim 2 For all i, j, k, s, each of the fields `shoot[`k`][`s`]`, `heal[`k`][`s`]`, `tag`, `num` and `pnum` in either a shared register $R_{i,j}$ or in user i's `from[`j`]`, is nondecreasing in time.

Proof. First consider the local variable `from[`i`]` of user i. Note that it is not changed by lines 7,8. The `num` and `shoot[`k`][`s`]` fields of this variable are only changed in lines 6,24 respectively, at which point they are incremented. The claim also holds for the field `pnum` of `from[`i`]`, as it is only changed by assignment from `num`. The `tag` of `from[`i`]` is only changed in lines 15,21. By the maximization step in line 13, and the increment of the timestamp in line 21, this field is nondecreasing as well.

Since $R_{i,j}$ is only changed by assigning `from[`i`]` to it in lines 7,18,25, the fields `shoot[`k`][`s`]`, `tag`, `num` and `pnum` of $R_{i,j}$ are also nondecreasing. The claim holds similarly for the above fields in the `from[`i`]` variable of other users, which is changed only by assignment from $R_{i,j}$ in line 8.

The above also establishes (by interchanging the roles of i and j) that, in particular, $R_{j,i}$`.shoot[`i`][`s`]` is nondecreasing. Note that lines 4 and 5 carry out the assignment `from[`i`].heal[`j`][`s`]` $:=$ $R_{j,i}$`.shoot[`i`][`s`]`, Since the field `heal[`k`][`s`]` of user i's `from[`i`]` is only changed here, with $j = k$, the claim holds this field as well. The same argument can be repeated for `heal[`k`][`s`]` in $R_{i,j}$ and in user j's `from[`i`]`. □

This claim shows that if

- ω is a subwrite of the value v_ω to one of the above fields of some subvariable, and
- ρ is a subread from the same subvariable, which obtains a value v_ρ for that field,

then $\omega \rightarrow \rho$ implies $v_\omega \leq v_\rho$, and conversely, $v_\rho < v_\omega$ implies $\rho \rightarrow \omega$.

The following notation will be used throughout the proofs.

Definition 5 Let $a \in \mathcal{A}$ be an action by user i, and `loc` a local variable of user i. Define `loc@`a as the value of that variable immediately after $f(a)$. Let $c \in \{e, h, r, t, p, w\}$ be a single character phase descriptor. Define `loc@`a`.c` as the value of that variable immediately after $f(a.c)$. If `loc` is a field of `from[`i`]` then the `from[`i`].` part is omitted.

Definition 6 A j-(sub)action/read/write is a (sub)action/read/write by user j.

The next definition provides a concise notation for the actions by other users which are the most important to a given action.

Definition 7 Let a be a non-aborting action by user i. Define a^j as the unique j-action b with $\mathtt{num@}b = \mathtt{from}[j]\mathtt{.pnum@}a\mathtt{.r}$. (For example, a^i is the last non-aborting i-action before a.)

Lemma 4 *Let a be a non-aborting action by user i and $b = a^j$. Define $js = \mathtt{from}[j]\mathtt{.ss@}a\mathtt{.r}$. Then*

1. *b does not abort*
2. *$js = \mathtt{ss@}b$*
3. *for all k, $0 \le k < n$, $\mathtt{from}[j]\mathtt{.heal}[k][js]\mathtt{@}a\mathtt{.r} = \mathtt{heal}[k][js]\mathtt{@}b$*
4. *$\mathtt{tag@}b\mathtt{.p} \le \mathtt{from}[j]\mathtt{.tag@}a\mathtt{.r} \le \mathtt{tag@}b$*
5. *for all k, s, $0 \le k < n$ and $s \in \{0,1\}$, $\mathtt{shoot}[k][s]\mathtt{@}b\mathtt{.p} \le \mathtt{from}[j]\mathtt{.shoot}[k][s]\mathtt{@}a\mathtt{.r} \le \mathtt{shoot}[k][s]\mathtt{@}b$*

Proof.

1. Consider the successive subwrites to $R_{j,i}$. A change in either of the fields pnum or ss of this variable must be the result of a $.\mathrm{p}_i$ subaction in line 18. Hence the numbers written to $R_{j,i}$.pnum are those of non-aborting actions, which proves that b doesn't abort.

2. Let c be the next non-aborting j-action following b. Then its subaction $c.\mathrm{p}_i$ complements $R_{j,i}$.ss, and increases $R_{j,i}$.pnum by an amount 1 greater than the number of aborting j-actions between b and c (which increment only .num). Moreover, no subaction between $b.\mathrm{p}_i$ and $c.\mathrm{p}_i$ changes these fields. Since, by claim 2, $\mathtt{from}[j]\mathtt{.pnum@}a\mathtt{.p} = \mathtt{num@}b < \mathtt{num@}c$, the order of the subactions must be

$$b.\mathrm{p}_i \to a.\mathrm{r}_j \to c.\mathrm{p}_i. \tag{1}$$

   Therefore, $js = \mathtt{from}[j]\mathtt{.ss@}a\mathtt{.p} = \mathtt{ss@}b$.

3. Note that j-actions following b up to and including c change only $\mathtt{from}[j]\mathtt{.heal}[k][1-js]$ and not $\mathtt{from}[j]\mathtt{.heal}[k][js]$. So the ordering of events (1) gives also $\mathtt{from}[j]\mathtt{.heal}[k][js]\mathtt{@}a\mathtt{.p} = \mathtt{heal}[k][js]\mathtt{@}b$.

4. From the first part of (1) and claim 2 we have $\mathtt{tag@}b\mathtt{.p} \le \mathtt{from}[j]\mathtt{.tag@}a\mathtt{.r}$. From the second part of (1) and the observation that $c.\mathrm{h}_i$ is the last subwrite to $R_{j,i}$ preceding $c.\mathrm{p}_i$, we obtain $\mathtt{from}[j]\mathtt{.tag@}a\mathtt{.r} \le \mathtt{tag@}c\mathtt{.h} = \mathtt{tag@}b$.

5. Similar to Item 4. From the first part of (1) and claim 2 follows $\texttt{shoot}[k][s]@b.\text{p} \leq \texttt{from}[j].\texttt{shoot}[k][s]@a.\text{r}$. From the second part of (1) and claim 2 follows $\texttt{from}[j].\texttt{shoot}[k][s]@a.\text{r} \leq \texttt{shoot}[k][s]@c.\text{p} = \texttt{shoot}[k][s]@b$.

□

4.2 Correctness of Algorithm 2

Lemma 5 *Let $\sigma = (\mathcal{A}, \rightarrow, \pi)$ be a system execution according to Algorithm 2. Then there is a function tag() satisfying (T1), (T2) and (T3).*

Proof. Let a be an action by user i. Define $tag : \mathcal{A} \rightarrow \mathbb{Q}$ as follows. If a doesn't abort, then simply set $tag(a) = \texttt{tag}@a$. Otherwise, if a aborts, let w be the non-aborting write of the value $\texttt{tmp.previous}@a$. Now set $tag(a) = tag(w)$ if a is a read, and set $tag(a) = tag(w) - \epsilon_a$, if a is a write, where $0 < \epsilon_a < 1$ is a fraction unique to a.

Claim 3 If an action a by user i aborts, then there exists a non-aborting write action w, such that $s(a) \rightarrow w \rightarrow f(a)$ with $\texttt{value}@w = \texttt{tmp.previous}@a$ and $\lceil tag(a) \rceil = tag(w)$.

Proof. Define $j = \texttt{j}@a$ as the user by which a sees itself wounded[3]. Let b be the j-action with $\texttt{num}@b = \texttt{tmp.num}@a$ and let $w = b^j$. Then $w \rightarrow s(b) \rightarrow f(a)$.

Let u be the j-action immediately preceding w. Then, with $is = \texttt{s}@a$,

$$\texttt{shoot}[i][is]@u \geq \texttt{shoot}[i][is]@w - 1 \geq \texttt{shoot}[i][is]@b - 2 \geq$$

$$\texttt{tmp.shoot}[i][is]@a - 2 \geq \texttt{heal}[j][is]@a + 1.$$

(Since a shoot counter increases by no more than the number of non-aborting write actions, the order $u \rightarrow w \rightarrow b$ of j-actions gives by claim 2 the first two inequalities. The definition of b gives the third inequality, and the fact that a aborts gives the fourth inequalty.) Hence, not $u \rightarrow a.\text{e}_j$, and consequently $s(a) \rightarrow f(u) \rightarrow w$.

Let c be the j-action immediately following b. By definition of b,

$$b.\text{h}_i \rightarrow a.\text{t}_j \rightarrow c.\text{h}_i.$$

Therefore, $\texttt{tmp.previous}@a = \texttt{value}@w$, and the definition of tag gives $\lceil tag(a) \rceil = tag(w)$. This completes the proof of claim 3. □

Claim 4 Let $a \rightarrow b$ be actions by users i and j respectively. If b is a non-aborting read, then

$$tag(b) = \texttt{tag}@b \geq \texttt{tag}@a,$$

[3] Since a aborts, it terminates execution of the protocol in line 11, with the local variable j as stated ($f(a) = a.\text{t}_j$).

and if b is a non-aborting write, then

$$tag(b) = \texttt{tag@}b \geq \texttt{tag@}a + 1.$$

Proof. Assume that b doesn't abort. Then, the definition of *tag* gives $tag(b) = \texttt{tag@}b$. If a aborts, then it writes $\texttt{tag@}a$ to $R_{i,j}$ in $a.\mathrm{h}_j$. Otherwise, a does so in $a.\mathrm{p}_j$ if it is a read, or in $a.\mathrm{w}_j$ if it is a write. Using $a \rightarrow b.\mathrm{r}_i$, claim 2 then gives $\texttt{from[max].tag@}b.\mathrm{p} \geq \texttt{from[}i\texttt{].tag@}b.\mathrm{r} \geq \texttt{tag@}a$. If b is a read, then $\texttt{tag@}b = \texttt{from[max].tag@}b.\mathrm{p}$, which proves the first case. If b is a write, then by line 21, $\texttt{tag@}b = (\texttt{from[max].tag@}b.\mathrm{p} \text{ div } n+1, j)$, proving the second case. □

We proceed with the proof of lemma 5.

(T1) states that $tag(r) = tag(\pi(r))$ must hold for all read actions r. This follows, in case r doesn't abort, directly from the definition of *tag*, and in case r aborts, from claim 3.

(T2) states that if $a \rightarrow b$ then $tag(a) \leq tag(b)$. Let $a \rightarrow b$ be actions by users i and j respectively. If a aborts, then by claim 3 there is a non-aborting w_a with $tag(a) \leq tag(w_a)$ and $f(w_a) \rightarrow f(a)$, in which case it suffices to prove (T2) for $w_a \rightarrow b$. Hence, without loss of generality, assume that a doesn't abort. If b aborts, then by claim 3 there is a non-aborting write action w_b with $a \rightarrow s(b) \rightarrow w_b$ and $tag(b) > tag(w_b) - 1$ which by claim 4 is at least $(tag(a) + 1) - 1$. Otherwise, (T2) follows directly from claim 4.

(T3) states that if $w_1 \neq w_2$ are both writes, then $tag(w_1) \neq tag(w_2)$. Let i, j be the users executing w_1 and w_2 respectively. If either w_1 or w_2 aborts, then either $tag(w_1)$ or $tag(w_2)$ has a unique fractional part, in which case (T3) holds. Now assume that neither one aborts. Then $tag(w_1) = i \pmod n$, whereas $tag(w_2) = j \pmod n$. Clearly, (T3) holds in case $i \neq j$. In case $i = j$, one write precedes the other, and (T3) follows from claim 4. □

5 Algorithm 3

The next transformation to Algorithm 3, see figure 4, consists of restricting the set of users from which to choose max to the *credible* users. The restriction implies that tags of actions which have received at least 5 shots from one user are removed from consideration.

```
12.  L = {j | ∀ k≠j from[k].shoot[j][from[j].ss]-from[j].heal[k][from[j].ss] < 5}
```

Figure 4: Algorithm 3; protocol for user i. Identical with figure 3, but for the exhibited line.

Lemma 6 *Let $\sigma = (\mathcal{A}, \rightarrow, \pi)$ be a system execution according to Algorithm 3. Then there is a function tag() satisfying (T1), (T2) and (T3).*

Proof. By lemma 5, it only needs to be shown that L as set in line 12 in Algorithm 2 can be restricted as in line 12 in Algorithm 3, without affecting the choice of max. It suffices to show claim 5, which means that an index associated with the maximum tag will not be eliminated by the restriction.

Claim 5 Let a be a non-aborting action by user i, and $j, k \in L, j \neq k$, be two indices. Define $js = \texttt{from}[j].\texttt{ss@}a.\text{r}$. If

$$\texttt{from}[k].\texttt{shoot}[j][js]\texttt{@}a.\text{r} - \texttt{from}[j].\texttt{heal}[k][js]\texttt{@}a.\text{r} \geq 5$$

(a sees j discredited by k), then

$$\texttt{from}[k].\texttt{tag@}a.\text{r} > \texttt{from}[j].\texttt{tag@}a.\text{r}$$

Proof. Let $b = a^j$. Using in succession item 5 of lemma 4, the assumption of the claim, and item 3 of lemma 4:

$$\texttt{shoot}[j][js]\texttt{@}a^k \geq \texttt{from}[k].\texttt{shoot}[j][js]\texttt{@}a.\text{r} \geq$$

$$\texttt{from}[j].\texttt{heal}[k][js]\texttt{@}a.\text{r} + 5 = \texttt{heal}[k][js]\texttt{@}b + 5.$$

This shows the existence of $w_0 \rightarrow \cdots \rightarrow w_5$, defined as follows. For $0 \leq m \leq 5$, let w_m be the *first* non-aborting write action by user k such that

$$\texttt{shoot}[j][js]\texttt{@}w_m = \texttt{heal}[k][js]\texttt{@}b + m.$$

Therefore, in particular, $w_5.\text{w}_i \rightarrow a.\text{r}_k$ so that, using in succession claim 2, claim 4, and line 21 of Algorithm 3:

$$\texttt{from}[k].\texttt{tag@}a.\text{r} \geq \texttt{tag@}w_5 \geq$$

$$(\texttt{tag@}w_4 \text{ div } n + 1, k) = (\texttt{from}[\texttt{max}].\texttt{tag@}w_4.\text{p} \text{ div } n + 2, k).$$

By item 4 of lemma 4, and lines 15, 21 of Algorithm 3:

$$\texttt{from}[j].\texttt{tag@}a.\text{r} \leq \texttt{tag@}b \leq (\texttt{from}[\texttt{max}].\texttt{tag@}b.\text{p} \text{ div } n + 1, j).$$

Thus, to prove the claim it suffices to show that

$$(\texttt{from}[\texttt{max}].\texttt{tag@}w_4.\text{p} \text{ div } n + 1, k) > (\texttt{from}[\texttt{max}].\texttt{tag@}b.\text{p} \text{ div } n, j). \quad (2)$$

Since by item 1 of lemma 4, b doesn't abort, it follows that $b.\text{r} \rightarrow b.\text{t}_k \rightarrow w_3.\text{w}_j \rightarrow w_4$ (by definition of the w_i's). The proof of (2) now depends on the order of j and k.

$k < j$ Let .pw be the first instance of a propagate or write phase in which user $m = \texttt{max@}b$ communicates $\texttt{from[max].tag@}b.\text{p}$ to user j. The ordering of the for loops in lines 18 and 25 implies the sequence of events

$$.\text{pw}_k \rightarrow .\text{pw}_j \rightarrow b.\text{r}_m \rightarrow w_4.\text{r}_m.$$

So

$$\texttt{from[max].tag@}w_4.\text{p} \geq \texttt{from[}m\texttt{].tag@}w_4.\text{r} \geq \texttt{from[max].tag@}b.\text{p}.$$

This proves (2).

$j < k$ Define w with $\texttt{tag@}w = \texttt{from[max].tag@}b.\text{p}$ as the write action by user $id = \texttt{from[max].tag@}b.\text{p} \bmod n$ that created the tag maximum among those read by b. The sequence of events

$$w.\text{p}_k \rightarrow w.\text{w}_j \rightarrow b.\text{r}_{\texttt{max@}b} \rightarrow w_4.\text{r}_{id}$$

shows that even if $\texttt{tag@}w$ is not visible to w_4 yet, in $w_4.\text{r}$ it obtains a tag not less than the tag propagated by $w.\text{p}$. Hence

$$\texttt{from[max].tag@}w_4.\text{p div } n+1 \geq \texttt{from[max].tag@}w.\text{p div } n+1 =$$

$$\texttt{tag@}w \text{ div } n = \texttt{from[max].tag@}b.\text{p div } n.$$

Together with $j < k$, this proves (2).

□

Claim 5 concludes the proof of lemma 6. □

6 Final Algorithm

It remains to get rid of the unbounded tags and counters in Algorithm 3. Additionally, the control-bit complexity can be minimized by removing from $R_{i,j}$ all information irrelevant to user j.

Due to the behaviour of shoot and heal counters, the perceived differences in lines 11 and 23 are in the range from 0 to 5 inclusive, since one of the comparands is owned by the user executing the operation. This is not the case in line 12 of figure 4. There it may be that neither of the two comparands are owned by the user executing the operation. Hence, they have been acquired by different atomic subactions spaced arbitrarily wide apart—which raises the possibility that the differences can rise unboundedly. It will be proved that this is not the case.

More precisely, suppose user i is executing the read phase of an action a. For $j < k$, other users (like j and k) may be executing arbitrarily many actions between $a.\text{r}_j$ and $a.\text{r}_k$. Usually, this has the effect that the values obtained from

$R_{k,i}$ in $a.\mathrm{r}_k$ are greater then they were at the time of $a.\mathrm{r}_j$. Thus, k appears to have higher tags, higher shoot counters, and higher heal counters. Since read actions do not introduce any values higher than those already in existence, this effect must be caused by write actions. However, a shooting mechanism has been introduced which makes actions abort if they overlap—in their read phase—a certain number of write actions by the same user. Furthermore, those aborting actions do not even get to line 12. Bounding the number of overlapped write actions also bounds the apparent increase in k's values. The next two sections discuss the differences between shoot and heal counters followed by a discussion of tag differences.

6.1 Upper Bound on Perceived Differences

Claim 6 Let a be a non-aborting action by user i, and $0 \leq j, k < n, j \neq k$. Define $js = \mathtt{from}[j].\mathtt{ss}@a.\mathrm{r}$. Then

$$\mathtt{from}[k].\mathtt{shoot}[j][js]@a.\mathrm{r} - \mathtt{from}[j].\mathtt{heal}[k][js]@a.\mathrm{r} \leq 8$$

Proof. We prove that a aborts if the above difference is more than 8. Let $b = a^j$ so that $js = \mathrm{s}@b$, by lemma 4, item 2. Thus, a difference of at least 9 gives, using in succession claim 2, the contradictory assumption, and lemma 4, item 3:

$$\mathtt{shoot}[j][js]@a^k \geq \mathtt{from}[k].\mathtt{shoot}[j][js]@a.\mathrm{r} \geq$$

$$\mathtt{from}[j].\mathtt{heal}[k][js]@a.\mathrm{r} + 9 = \mathtt{heal}[k][js]@b + 9.$$

This establishes the existence of $w_0 \rightarrow \cdots \rightarrow w_9$, defined as follows. For $0 \leq m \leq 9$, let w_m be the *first* non-aborting write action by user k such that

$$\mathtt{shoot}[j][js]@w_m = \mathtt{heal}[k][js]@b + m.$$

Because a shoot counter of a user is at most 5 ahead of the corresponding heal counter, and by definition of w_6:

$$\mathtt{from}[j].\mathtt{heal}[k][js]@w_6 \geq \mathtt{shoot}[j][js]@w_6 - 5 =$$

$$\mathtt{heal}[k][js]@b + 1 = \mathtt{heal}[k][js]@c + 1,$$

where c is the next non-aborting j-action after b (lines 2 and 16 show that non-aborting actions use the two sets of heal counters alternately). This implies $\neg(w_6.\mathrm{r}_j \rightarrow c.\mathrm{p}_i)$. Using $a^j = b \rightarrow c$ for the second arrow, we get

$$a.\mathrm{h} \rightarrow a.\mathrm{r}_j \rightarrow c.\mathrm{p}_i \rightarrow w_6.\mathrm{r}_j \rightarrow w_7.$$

As a consequence (with $is = \mathrm{s}@a$), by claim 2,

$$\mathtt{from}[i].\mathtt{heal}[k][is]@w_7 \geq \mathtt{heal}[k][is]@a,$$

so with w_7, w_8 and w_9 shooting at user i,

$$\texttt{shoot}[i]\,[is]\,@w_9 \geq \texttt{heal}[k]\,[is]\,@a + 3.$$

By definition of w_9, $w_9.\mathrm{w}_i \rightarrow a.\mathrm{r}_k \rightarrow a.\mathrm{t}_k$, hence a aborts. □

6.2 Lower Bound on Perceived Differences

Again consider the case that user k shoots at j, but now $k < j$, hence $a.\mathrm{r}_k \rightarrow a.\mathrm{r}_j$. It is possible that between these two atomic reads, user k performs some write actions, and user j updates its heal counters in a read action. It can then be the case that the heal counter read in $a.\mathrm{r}_j$ is actually greater than the shoot counter read in $a.\mathrm{r}_k$!

Claim 7 Let a be a non-aborting action by user i, and $0 \leq j, k < n, j \neq k$. Define $js = \texttt{from}[j]\,.\,\mathrm{ss}@a.\mathrm{r}$. Then

$$\texttt{from}[k]\,.\,\texttt{shoot}[j]\,[js]\,@a.\mathrm{r} - \texttt{from}[j]\,.\,\texttt{heal}[k]\,[js]\,@a.\mathrm{r} \geq -4$$

Proof. We prove that a aborts if the above difference is less than -4. Let $b = a^j$ so that $js = \mathrm{s}@b$, by lemma 4, item 2. A difference of at most -5 gives, using in succession item 5 of lemma 4 claim 2, item 3 of lemma 4, and the contradictory assumption:

$$\texttt{shoot}[j]\,[js]\,@b^k \geq \texttt{from}[k]\,.\,\texttt{shoot}[j]\,[js]\,@b.\mathrm{r} \geq \texttt{heal}[k]\,[js]\,@b =$$

$$\texttt{from}[j]\,.\,\texttt{heal}[k]\,[js]\,@a.\mathrm{r} \geq \texttt{from}[k]\,.\,\texttt{shoot}[j]\,[js]\,@a.\mathrm{r} + 5.$$

This establishes the existence of $w_0 \rightarrow \ldots \rightarrow w_5$, defined as follows. For $0 \leq m \leq 5$, let w_m be the *first* non-aborting write action by user k such that

$$\texttt{shoot}[j]\,[js]\,@w_m = \texttt{from}[k]\,.\,\texttt{shoot}[j]\,[js]\,@a.\mathrm{r} + m.$$

Note that with this definition, $b.\mathrm{e}_k \rightarrow w_5.\mathrm{w}_j$ would imply $\texttt{heal}[k]\,[js]\,@b < \texttt{shoot}[j]\,[js]\,@w_5$, contradictory to the derivation above.

By definition of w_1, w_2,

$$a.\mathrm{h} \rightarrow a.\mathrm{r}_k \rightarrow w_1.\mathrm{w}_i \rightarrow w_2.$$

As a consequence (with $is = \mathrm{s}@a$), by claim 2,

$$\texttt{from}[i]\,.\,\texttt{heal}[k]\,[is]\,@w_2 \geq \texttt{heal}[k]\,[is]\,@a,$$

so with w_2, w_3 and w_4 shooting at user i,

$$\texttt{shoot}[i]\,[is]\,@w_4 \geq \texttt{heal}[k]\,[is]\,@a + 3.$$

But by definition of w_5 and the definition of b, the order of events is

$$w_4.\mathrm{w}_i \rightarrow w_5.\mathrm{w}_j \rightarrow b.\mathrm{e}_k \rightarrow b.\mathrm{p}_i \rightarrow a.\mathrm{r}_j \rightarrow a.\mathrm{t}_k,$$

and hence a aborts. □

The conclusion is that the result of any comparison between a shoot counter and the corresponding heal counter always lies in the range $-4, \ldots, 8$ inclusive. Since the only purpose served by these counters is comparison, they can be stored and used modulo $8-(-4)+1 = 13$. For the algorithm, this only involves performing the shoot counter increment modulo 13 and mapping all differences between shoot and heal counters to the above range. This mapping can be done in the comparison by treating $9, 10, 11, 12$ as $-4, -3, -2, -1$. The name can be changed accordingly from counter to *ring*, resulting in 'shoot rings' and 'heal rings'. In the next section we show that the tags can be turned into such a ring as well.

6.3 Range of Credible Timestamps

Having restricted L in Algorithm 3 to the set of users whose actions are not discredited, one would expect the tags—and therefore the timestamps—to be relatively close to each other. This is formalized in the next claim.

Claim 8 Let a be a non-aborting action by user i, and $0 \leq j, l < n$. Let $b = a^j$ and define $js = \mathtt{s@}b = \mathtt{from}[j]\mathtt{.ss@}a.\mathrm{r}$. If

$$\mathtt{from}[l]\mathtt{.tag@}a.\mathrm{r} - \mathtt{from}[j]\mathtt{.tag@}a.\mathrm{r} > 9n^2 + n - 1$$

then there exists a $k, 0 \leq k < n$ such that

$$\mathtt{from}[k]\mathtt{.shoot}[j][js]\mathtt{@}a.\mathrm{r} - \mathtt{from}[j]\mathtt{.heal}[k][js]\mathtt{@}a.\mathrm{r} \geq 5.$$

Proof. Division by n yields

$$\mathtt{from}[l]\mathtt{.tag@}a.\mathrm{r} \text{ div } n - \mathtt{from}[j]\mathtt{.tag@}a.\mathrm{r} \text{ div } n > 9n,$$

which means that more than $9n$ non-aborting write actions that start before $a.\mathrm{r}_l$ have a timestamp greater than $\mathtt{from}[j]\mathtt{.tag@}a.\mathrm{r}$ div n, which, by lemma 4 item 4 and lines 15, 21, is at least $\mathtt{from[max].tag@}b.\mathrm{p}$ div n. Therefore none of these actions precede $b.\mathrm{r}$. Since each user can start at most one such action before $b.\mathrm{r}$, $b.\mathrm{h}$ precedes more than $8n$ such write actions. Let k be a user which executes at least 9 of these write actions, $b.\mathrm{h} \rightarrow w_1 \rightarrow \cdots \rightarrow w_9$. The sequence of events

$$w_6.\mathrm{r}_i \rightarrow w_7 \rightarrow w_8 \rightarrow s(w_9) \rightarrow a.\mathrm{r}_l \rightarrow a.\mathrm{t}_k,$$

and the fact that a would abort if $a.\mathrm{h}_k \rightarrow w_6.\mathrm{r}_i$, imply that

$$w_5.\mathrm{w}_i \rightarrow w_6.\mathrm{r}_i \rightarrow a.\mathrm{h}_k \rightarrow a.\mathrm{r}_k. \tag{3}$$

If $k = j$, then $b \rightarrow w_5 \rightarrow a.\mathrm{r}_j$, which contradicts $b = a^j$. Hence, $k \neq j$. Since $b.\mathrm{h} \rightarrow w_1$, by claim 2,

$$\texttt{from}[j].\texttt{heal}[k][js]@w_1 \geq \texttt{heal}[k][js]@b.$$

Therefore, with w_1 through w_5 shooting at user j,

$$\texttt{shoot}[j][js]@w_5 \geq \texttt{heal}[k][js]@b + 5. \tag{4}$$

Applying in succession (3) in combination with claim 2, (4), and lemma 4, item 3:

$$\texttt{from}[k].\texttt{shoot}[j][js]@a.\mathrm{r} \geq \texttt{shoot}[j][js]@w_5 \geq$$

$$\texttt{heal}[k][js]@b + 5 = \texttt{from}[j].\texttt{heal}[k][js]@a.\mathrm{r} + 5.$$

□

By claim 8, the result of comparing two alive tags in Algorithm 3 lies in the range $-n(9n+1)+1, \ldots, n(9n+1)-1$ inclusive. That being the only use of tags, reduce the tags modulo $2n(9n+1)$, without affecting the behaviour of the algorithm (using a multiple of n to accomodate the index). With these *tag rings*, all variables used by the algorithm have been bounded. The resulting algorithm, Algorithm 4, incorporating all three types of ring, is depicted in figure 5.

Proof of theorem 1. Algorithm 4 is obviously wait-free. Let $\sigma = (\mathcal{A}, \rightarrow, \pi)$ be a system execution according to Algorithm 4. Condition (S1) is trivially satisfied. By lemma 6, claim 6, claim 7, and claim 8, there is a function *tag* satisfying (T1), (T2), and (T3). Therefore, Algorithm 4 implements an atomic variable, by lemma 2. It uses n^2 atomic wait-free 1-reader 1-writer variables (the $R_{i,j}$'s). The protocol uses $O(n)$ of accesses of subvariables per read/write, and $O(n)$ control bits per subvariable (a precise analysis is given in section 8). □

6.4 Minimizing Redundancy

We actually need only $n(n-1)$ subvariables since the diagonal subvariables $R_{i,i}$ and their accesses are superfluous and can be omitted. Since no purpose is served by shooting at oneself, we can also remove all suicide-related shoot and heal rings. Note furthermore that user j makes no use of the fields $\texttt{heal}[\neq j]$[1-ss] in $R_{i,j}$. Thus, in addition to the element $\texttt{heal}[j]$[1-ss], only a single heal[] array needs to be stored in $R_{i,j}$, provided the second index is understood to be $R_{i,j}.\mathrm{ss}$. Table 1 shows all the fields in $R_{3,1}$ in the case of 5 users.

```
1.  from[i].previous := from[i].value
2.  s := 1 - from[i].ss
3.  for j:=0 to n-1 do
4.    Read tmp := R_{j,i}
5.    from[i].heal[j][s] := tmp.shoot[i][s]
6.  for j:=0 to n-1 do Write R_{i,j} := from[i]
7.  for j:=0 to n-1 do Read from[j] := R_{j,i}
8.  for j:=0 to n-1 do
9.    Read tmp := R_{j,i}
10.   if (tmp.shoot[i][s] - from[i].heal[j][s]) mod 13 ≥ 3
11.   then return tmp.previous
12. L := { j | ∀ k≠j (from[k].shoot[j][from[j].ss]
      - from[j].heal[k][from[j].ss]) mod 13 ∉ {5,...,8} }
13. select max ∈ L such that ∀ j ∈ L
14.   (from[max].tag - from[j].tag) mod 2n(9n+1) < n(9n+1)
15. from[i].value := from[max].value
16. from[i].tag := from[max].tag
17. from[i].ss := s
18. for j:=0 to n-1 do Write R_{i,j} := from[i]
19. if read_action then return from[i].value
20. from[i].value := newvalue
21. from[i].tag := (n * (from[max].tag div n + 1) + i) mod 2n(9n+1)
22. for j:=0 to n-1 and s ∈ {0,1} do
23.   if (from[i].shoot[j][s] - from[j].heal[i][s]) mod 13 < 5
24.   then from[i].shoot[j][s] := (from[i].shoot[j][s]+1) mod 13
25. for j:=0 to n-1 do Write R_{i,j} := from[i]
```

Figure 5: Final Algorithm (4); protocol for user i

shoot[0][0] shoot[0][1]	shoot[1][0] shoot[1][1]	shoot[2][0] shoot[2][1]	shoot[4][0] shoot[4][1]
value	tag	ss	previous
heal[0][ss]	heal[1][1-ss] heal[1][ss]	heal[2][ss]	heal[4][ss]

Table 1: the fields of $R_{3,1}$ with 5 users

7 Subproblems and Implementation

The final algorithm implements a multi-user variable with single-reader variables. Using *multi-reader* variables as the basis of a multi-user implementation simplifies matters. The interested reader is referred to appendix A. Another projection of the algorithm implements a multi-reader variable from single-reader ones. See appendix B for details.

Some of the algorithms presented here have been implemented and tested. See appendix C for the results.

8 Conclusions

It has now been proven that Algorithm 4, by equivalence with Algorithm 2, correctly implements an atomic multi-user variable without waiting. Analysing the complexity (indicating the optimized version of section 6.4 in parentheses) yields the following. The number of subvariable accesses in an operation execution is at most $6n$ (optimized $6(n-1)$) in the case of a write action and at most $5n$ (optimized $5(n-1)$) in the case of a read action. This proves the $O(n)$ subvariable accesses or "time complexity" of theorem 1. The number of bits in each subvariable $R_{i,j}$ equals $2b + 4n \log 13 + \log(2n(9n+1)) + 1$, (optimized $2b + (3n-2)\log 13 + \log(2n(9n+1)) + 1$), with b the number of bits ("width") of the shared variable V. This proves the $O(n)$ control bits or "space complexity" of theorem 1, and thereby completes its proof. Unfortunately, the extra value `previous` may put a heavy burden on the size of the subvariables if $b >> n$. This cannot be helped since the whole solution relies on the principle of abortion and aborting actions are required to return the value of a completed write.

We would like to make some final remarks about the possibility of parallellizing the algorithm. The order of the for loops in Algorithm 4 is only important in lines 18 and 25. All other for loops could be replaced by a "`for all j in`" construct which means that the different instances of the loop body can be executed in parallel. By increasing the lethal number of shots from 5 to 6, the above two loops can also be relaxed. This would enlarge the range of perceived `shoot - heal` differences to $-4, \ldots, 9$ hence a 14-valued shoot/heal ring (instead of 13). The size of the tag ring should accordingly be increased from $2n(9n+1)$ to $2n(10n+1)$. These numbers follow from careful examination of the claims 6, 7, and 8 in the light of the incremented lethal number of shots. Hence, a parallel version of the algorithm can be made to run in $O(1)$ time complexity and the same ($O(n)$) space complexity.

9 Acknowledgements

We thank Amos Israeli and Leslie Lamport for fruitful discussions.

A From Multi-reader to Multi-user

A restriction of our original goal corresponds to the problem that [17] and [12] unsuccesfully tried to solve. Viz., to implement an atomic wait-free n-user shared variable V using atomic multi-reader variables $R_i (0 \leq i < n)$ for which user i is the single writer and the other $n-1$ users are the readers.

To implement V, collapse row $R_{i,0}, \ldots, R_{i,n-1}$ to the single multi-reader variable R_i owned by user i. So the write loops reduce to a single atomic write. Each variable $R_{j,i}$ read is replaced by R_j. The previous field is no longer needed, since the value of a write action becomes visible to all users at the same time, thus making it suitable for return by an aborting action. The complete propagate phase can be skipped for similar reasons. The wounding number of shots can be reduced from 3 to 2, since the write action firing the second shot must have finished by the time the shot is noticed. The lethal (discrediting) number of shots can then be reduced to 4. The affected ring-sizes are 9 for the shoot/heal rings and $2n(6n+1)$ for the tag rings. Figure 6 shows the resulting algorithm, given without a proof of correctness. This solution uses $O(n)$ control bits per variable R_i and $O(n)$ atomic accesses of subvariables per read or write action on V.

```
 1. s := 1 - from[i].ss
 2. for j:=0 to n-1 do
 3.   Read tmp := R_j
 4.   from[i].heal[j][s] := tmp.shoot[i][s]
 5. Write R_i := from[i]
 6. for j:=0 to n-1 do Read from[j] := R_j
 7. for j:=0 to n-1 do
 8.   Read tmp := R_j
 9.   if (tmp.shoot[i][s] - from[i].heal[j][s]) mod 9 ≥ 2
10.   then return tmp.value
11. L := { j | ∀ k≠j (from[k].shoot[j][from[j].ss]
    - from[j].heal[k][from[j].ss]) mod 9 ∉ {4,5,6} }
12. select max ∈ L such that ∀ j ∈ L
13.   (from[max].tag - from[j].tag) mod 2n(6n+1) < n(6n+1)
14. if read_action then return from[max].value
15. from[i].value := newvalue
16. from[i].tag := (n*(from[max].tag div n + 1) + i) mod 2n(6n+1)
17. from[i].ss := s
18. for j:=0 to n-1 and s ∈ {0,1} do
19.   if (from[i].shoot[j][s] - from[j].heal[i][s]) mod 9 < 4
20.   then from[i].shoot[j][s] := (from[i].shoot[j][s]+1) mod 9
21. Write R_i := from[i]
```

Figure 6: Algorithm 5; protocol for user i

B From Single-reader to Multi-reader

Another restriction of our original goal above corresponds to the problem that was attacked in [15, 7, 3, 11, 6]. Viz., to implement an atomic wait-free multi-reader shared variable V with n readers $0, \ldots, n-1$ and a single writer n. V is implemented using atomic single-reader variables $R_{i,j}, (0 \leq i, j \leq n)$ for which user i is the single associated writer and user j is the single associated reader. We show how to implement V using $O(n)$ control bits in each variable $R_{n,j}$ owned by the writer, and $O(1)$ control bits in each variable $R_{i,j}, (0 \leq i < n)$ owned by the readers.

With a single writer, tags equal timestamps, none of the readers need a shoot array, and the `heal[j][s]` array of a reader collapses into `heal[s]`. Also the writer doesn't need a heal array and therefore no ss. While the wounding number of shots remains 3, the discrediting number of shots can be reduced to 3. Because the writer will be last in the reader's read phase, the observed heal counters (timestamps) are at most 1 greater than the corresponding shoot counters (resp. timestamp) of the writer. As a result, the size of the shoot/heal rings can be reduced to 8. Analysis also shows that alive readers are perceived to be at most 6 tags behind the writer, so a tag ring of size 8 suffices. There are separate protocols for the writer (fig 7) and the readers (fig 8), as to reflect the functional difference. Note that the writer has no propagate phase—there are no other writers that could have more recent values.

```
1. from.previous := from.value
2. from.value := newvalue
3. from.tag := (from.tag+1) mod 8
4. for j:=0 to n-1 do
5.   Read tmp := R_{j,n}
6.   for s ∈ {0,1} do
7.   if (from.shoot[j][s] - tmp.heal[s]) mod 8 < 3
8.   then from.shoot[j][s] := (from.shoot[j][s]+1) mod 8
9. for j:=0 to n-1 do Write R_{n,j} := from
```

Figure 7: Algorithm 6; protocol for writer (user n)

```
 1. s := 1 - from[i].ss
 2. Read from[n] := R_{n,i}
 3. from[i].heal[s] := from[n].shoot[i][s]
 4. Write R_{i,n} := from[i]
 5. for j:=0 to n do Read from[j] := R_{j,i}
 6. if (from[n].shoot[i][s] - from[i].heal[s]) mod 8 ≥ 3
 7. then return from[n].previous
 8. max := n
 9. for j:=0 to n-1 do
10.   if from[j].tag = (from[n].tag+1) mod 8 ∧
11.   (from[n].shoot[j][from[j].ss]-from[j].heal[from[j].ss]) mod 8 ∉ {3,...,6}
12.   then max := j
13. from[i].value := from[max].value
14. from[i].tag := from[max].tag
15. from[i].ss := s
16. for j:=0 to n-1 do Write R_{i,j} := from[i]
17. return from[i].value
```

Figure 8: Algorithm 6; protocol for reader i

C Simulation of the Algorithm

Both Algorithm 2 and Algorithm 4 have been implemented and a program was written which simulates (pseudo-) randomly interleaved system executions of both algorithms in parallel.

An explanation of the interleaving process follows. First a distribution of *sleeptimes* was fixed. A sleeptime is a number of steps which a user has to wait before it may continue the execution of a protocol. Initially all users start *awake*, i.e., with a sleeptime of zero. In general, if more than one user is awake, then one such user is selected at random and a new, positive sleeptime is chosen from the sleeptimes distribution. This procedure is repeated until a single user remains awake. In that case the minimum of the sleeptimes of the other users is determined and taken as the number of steps to run the remaining user. A step is defined as a part of the protocol involving exactly one primitive register access and any local computations—this reflects the atomicity of the constituent registers. When the desired number of steps is completed, it is subtracted from the sleeptimes of the other users and the whole process repeats. This sleeptime algorithm helps to find counterexamples in which one user is required to sleep while most of the other ones repeatedly run.

By testing whether actions have the same behaviour under both protocols, the program empirically tests the correctness of Algorithm 4 relative to Algorithm 2. While the latter uses unbounded counters and tags, the former uses modulos and a restricted "alive" set to choose `max` from. The simulator allows adjustment of the following parameters:

- number of users (n)
- lower bound of the range of shoot/heal ring values (lb)
- upper bound of the range of shoot/heal ring values (ub)
- number of shots to wound an action (wd)
- number of shots to discredit an action (kl)
- size of the tag ring (ts)
- percentage of write actions (wp)
- number of steps simulated with same sleeptime distribution (nd)
- size of the sleeptime distribution (ds)
- maximum sleeptime (ms)

Given the last two parameters, the logarithms of the ds sleeptimes were chosen randomly from the uniform distribution $[0, \log ms]$. After each nd simulated

steps a new sleeptime distribution is chosen. Sooner or later a distribution will be found which is appropriate for the search of a counterexample. This means that nd should be taken neither too small nor too large; any value between 10^3 and 10^6 seems reasonable. Curiously, the best value of wp for finding counterexamples proved to be 100. Furthermore, the set of counterexamples found with $n = 4$ was a proper subset of those found with $n = 3$.

The counterexamples which can be shown to exist for $lb = -3$ and $ub = 7$ proved to be too hard to find for the simulator within some 10^7 simulated steps. The bounds of the shoot/heal rings are optimal in the sense that they are sufficient to preserve correctness of the algorithm, while tightening either of them gives rise to a counterexample. The size of the tag ring is larger than necessary, but optimal bounds (like $n(18n - 27)$) are hard to obtain and much more so to prove.

While the use of a simulator may seem of questionable value in supporting the correctness proofs of the algorithms, it has proven to be of great assistance in the development of both the algorithms and their formal proofs. Several alternative implementations of the shooting construct have been tried out, and some were refuted by the simulator much faster than could have been done manually.

References

[1] B. Awerbuch, L. Kirousis, E. Kranakis, P.M.B. Vitányi, *A proof technique for register atomicity*, Proc. 8th Conference on Foundations of Software Technology & Theoretical Computer Science, Lecture Notes in Computer Science, vol. 338, pp. 286–303, Springer Verlag, 1988.

[2] B. Bloom, *Constructing Two-writer Atomic Registers*, IEEE Transactions on Computers, vol. 37, pp. 1506–1514, 1988.

[3] J.E. Burns and G.L. Peterson, *Constructing Multi-reader Atomic Values From Nonatomic Values*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 222–231, 1987.

[4] D. Dolev and N. Shavit, *Bounded Concurrent Time-Stamp Systems Are Constructible*, Proc. 21th ACM Symposium on Theory of Computing, pp. 454–466, 1989.

[5] M.P. Herlihy, *Impossibility and Universality Results for Wait-Free Synchronization*, Proc. 7th ACM Symposium on Principles of Distributed Computing, 1988.

[6] A. Israeli and M. Li, *Bounded Time-Stamps*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 371–382, 1987.

[7] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *Atomic Multireader Register*, Proc. 2nd International Workshop on Distributed Computing, Amsterdam, J. van Leeuwen (ed.), Springer Verlag Lecture Notes in Computer Science, vol. 312, pp. 278–296, July 1987.

[8] L. Lamport, *On Interprocess Communication Parts I and II*, Distributed Computing, vol. 1, pp. 77–101, 1986.

[9] M. Li and P.M.B. Vitányi, *A very simple construction for atomic multi-writer register*, Techn. Report TR 01-87, Aiken Comp. Lab., Harvard University, November 1987. Also, pp. 488-505 in: *Proc. International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989.

[10] N. Lynch and M. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, Proc. 6th ACM Symposium on Principles of Distributed Computing, 1987.

[11] R. Newman-Wolfe, *A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 232–248, 1987.

[12] G.L. Peterson and J.E. Burns, *Concurrent reading while writing II: the multiwriter case*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 383–392, 1987.

[13] G.L. Peterson, *Concurrent reading while writing*, ACM Transactions on Programming Languages and Systems, vol. 5, No. 1, pp. 46–55, 1983.

[14] R. Schaffer, *On the correctness of atomic multi-writer registers*, Technical Report MIT/LCS/TM-364, MIT lab. for Computer Science, June 1988.

[15] A.K. Singh, J.H. Anderson, M.G. Gouda, *The Elusive Atomic Register Revisited*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 206–221, 1987.

[16] K. Vidyasankar, *Converting Lamport's Regular Register to an atomic register*, Information Processing Letters, vol. 28, pp. 287–290, 1988.

[17] P.M.B. Vitányi, B. Awerbuch, *Atomic Shared Register Access by Asynchronous Hardware*, Proc. 27th IEEE Symposium on Foundations of Computer Science, pp. 233–243, 1986. (Errata, Ibid.,1987)

The ITLI Prepublication Series

1990

Logic, Semantics and Philosophy of Language

LP-90-01 Jaap van der Does — A Generalized Quantifier Logic for Naked Infinitives
LP-90-02 Jeroen Groenendijk, Martin Stokhof — Dynamic Montague Grammar
LP-90-03 Renate Bartsch — Concept Formation and Concept Composition
LP-90-04 Aarne Ranta — Intuitionistic Categorial Grammar
LP-90-05 Patrick Blackburn — Nominal Tense Logic
LP-90-06 Gennaro Chierchia — The Variablity of Impersonal Subjects
LP-90-07 Gennaro Chierchia — Anaphora and Dynamic Logic
LP-90-08 Herman Hendriks — Flexible Montague Grammar
LP-90-09 Paul Dekker — The Scope of Negation in Discourse, towards a flexible dynamic Montague grammar
LP-90-10 Theo M.V. Janssen — Models for Discourse Markers
LP-90-11 Johan van Benthem — General Dynamics
LP-90-12 Serge Lapierre — A Functional Partial Semantics for Intensional Logic
LP-90-13 Zhisheng Huang — Logics for Belief Dependence
LP-90-14 Jeroen Groenendijk, Martin Stokhof — Two Theories of Dynamic Semantics
LP-90-15 Maarten de Rijke — The Modal Logic of Inequality
LP-90-16 Zhisheng Huang, Karen Kwast — Awareness, Negation and Logical Omniscience
LP-90-17 Paul Dekker — Existential Disclosure, Implicit Arguments in Dynamic Semantics

Mathematical Logic and Foundations

ML-90-01 Harold Schellinx — Isomorphisms and Non-Isomorphisms of Graph Models
ML-90-02 Jaap van Oosten — A Semantical Proof of De Jongh's Theorem
ML-90-03 Yde Venema — Relational Games
ML-90-04 Maarten de Rijke — Unary Interpretability Logic
ML-90-05 Domenico Zambella — Sequences with Simple Initial Segments
ML-90-06 Jaap van Oosten — Extension of Lifschitz' Realizability to Higher Order Arithmetic, and a Solution to a Problem of F. Richman
ML-90-07 Maarten de Rijke — A Note on the Interpretability Logic of Finitely Axiomatized Theories
ML-90-08 Harold Schellinx — Some Syntactical Observations on Linear Logic
ML-90-09 Dick de Jongh, Duccio Pianigiani — Solution of a Problem of David Guaspari
ML-90-10 Michiel van Lambalgen — Randomness in Set Theory
ML-90-11 Paul C. Gilmore — The Consistency of an Extended NaDSet

Computation and Complexity Theory

CT-90-01 John Tromp, Peter van Emde Boas — Associative Storage Modification Machines
CT-90-02 Sieger van Denneheuvel, Gerard R. Renardel de Lavalette — A Normal Form for PCSJ Expressions
CT-90-03 Ricard Gavaldà, Leen Torenvliet, Osamu Watanabe, José L. Balcázar — Generalized Kolmogorov Complexity in Relativized Separations
CT-90-04 Harry Buhrman, Edith Spaan, Leen Torenvliet — Bounded Reductions
CT-90-05 Sieger van Denneheuvel, Karen Kwast — Efficient Normalization of Database and Constraint Expressions
CT-90-06 Michiel Smid, Peter van Emde Boas — Dynamic Data Structures on Multiple Storage Media, a Tutorial
CT-90-07 Kees Doets — Greatest Fixed Points of Logic Programs
CT-90-08 Fred de Geus, Ernest Rotterdam, Sieger van Denneheuvel, Peter van Emde Boas — Physiological Modelling using RL
CT-90-09 Roel de Vrijer — Unique Normal Forms for Combinatory Logic with Parallel Conditional, a case study in conditional rewriting

Other Prepublications

X-90-01 A.S. Troelstra — Remarks on Intuitionism and the Philosophy of Mathematics, Revised Version
X-90-02 Maarten de Rijke — Some Chapters on Interpretability Logic
X-90-03 L.D. Beklemishev — On the Complexity of Arithmetical Interpretations of Modal Formulae
X-90-04 — Annual Report 1989
X-90-05 Valentin Shehtman — Derived Sets in Euclidean Spaces and Modal Logic
X-90-06 Valentin Goranko, Solomon Passy — Using the Universal Modality: Gains and Questions
X-90-07 V.Yu. Shavrukov — The Lindenbaum Fixed Point Algebra is Undecidable
X-90-08 L.D. Beklemishev — Provability Logics for Natural Turing Progressions of Arithmetical Theories
X-90-09 V.Yu. Shavrukov — On Rosser's Provability Predicate
X-90-10 Sieger van Denneheuvel, Peter van Emde Boas — An Overview of the Rule Language RL/1
X-90-11 Alessandra Carbone — Provable Fixed points in $I\Delta_0+\Omega_1$, revised version
X-90-12 Maarten de Rijke — Bi-Unary Interpretability Logic
X-90-13 K.N. Ignatiev — Dzhaparidze's Polymodal Logic: Arithmetical Completeness, Fixed Point Property, Craig's Property
X-90-14 L.A. Chagrova — Undecidable Problems in Correspondence Theory
X-90-15 A.S. Troelstra — Lectures on Linear Logic

1991

Mathematical Logic and Foundations

ML-91-01 Yde Venema — Cylindric Modal Logic
ML-91-02 Alessandro Berarducci, Rineke Verbrugge — On the Metamathematics of Weak Theories
ML-91-03 Domenico Zambella — On the Proofs of Arithmetical Completeness for Interpretability Logic

Computation and Complexity Theory

CT-91-01 Min Li, Paul M.B. Vitanyi — Kolmogorov Complexity Arguments in Combinatorics
CT-91-02 Min Li, John Tromp, Paul M.B. Vitanyi — How to Share Concurrent Wait-Free Variables
CT-91-03 Min Li, Paul M.B. Vitanyi — Average Case Complexity under the Universal Distribution Equals Worst Case Complexity

Other Prepublications

X-91-01 Alexander Chagrov, Michael Zakharyaschev — The Disjunction Property of Intermediate Propositional Logics
X-91-02 Alexander Chagrov, Michael Zakharyaschev — On the Undecidability of the Disjunction Property of Intermediate Propositional Logics
X-91-03 V. Yu. Shavrukov — Subalgebras of Diagonizable Algebras of Theories containing Arithmetic
X-91-04 K.N. Ignatiev — Partial Conservativity and Modal Logics
X-91-05 Johan van Benthem — Temporal Logic