Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

Ming Li, P.M.B. Vitanyi

A very simple construction for atomic multiwriter register (Extended abstract)

Computer Science/Department of Algorithmics & Architecture Report CS-R8751 November

A Very Simple Construction for Atomic Multiwriter Register
(Extended Abstract)

Ming Li

Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138

Paul M.B. Vitanyi

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

ABSTRACT

This paper introduces a new and conceptually very simple algorithm to implement an atomic n-reader n-writer variable directly from atomic 1-reader 1-writer variables, using bounded tags. The algorithm is developed top-down from the unbounded tag method in [VA]. This is the first direct such construction, and considerably improves the complexity of all known compound constructions. The algorithm uses new techniques, but its main virtue is that it is *conceptually very simple and easily proved correct.*

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.
Note: This paper is submitted for publication elsewhere.

1. Introduction

A shared variable is *atomic*, if each read and write of it actually happens, or can be thought to happen, in an indivisible instant of time, irrespective of its actual duration. (The time-instant in which the action is seemingly executed must be in between the beginning and the end of the actual duration.) We construct an atomic bounded tag n-reader n-writer shared variable (or *register*) from atomic 1-reader 1-writer shared variables. The construction is top-down from the Vitanyi-Awerbuch unbounded tag algorithm [VA].

Theorem 1. *An atomic n-reader n-writer variable can be constructed from atomic* 1-*reader 1-writer variables, using $5n$ accesses of subvariables per operation execution and $O(n)$ control bits per subvariable.*

This is the first direct bounded-tag construction of atomic n-reader n-writer variable from atomic 1-reader 1-writer variables, and considerably improves the complexity of any known compound construction. Worst case complexity comparison of the direct solution here with the best

The work of the first author was supported in part by the National Science Foundation under Grants DCR-8301766, DCR-8606366, and Office of Naval Research Grant N00014-85-K-0445.

Report CS-R8751
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

combinations:

paper	control bits	atomic accesses
[This paper]	$O(n^3)$	$5n$
[BP]+[PB]	$\Omega(n^3)$	$\Omega(n^3)$
[IL]	$\Omega(n^4)$	$\Omega(n^2)$
[SAG,KKV,BP,NW]+[IL]	$\Omega(n^3)$	$\Omega(n^2)$

The algorithm uses new techniques. We want to stress, that we view as the most important contribution that this is the first such algorithm *that is conceptually very simple and easily proved correct.* It provides certainty in a troubled field. Related work is [Pe], [La], [B], [VA], [AGS], [Ly], [KKV], [PB], [BP], [AKKV], [IL], [NW]. It is extremely hard to understand the proposed algorithms, or the correctness proofs. In fact, with exception of [Pe], [La], the 2-writer Bloom algorithm [B] and the unbounded tag Vitanyi-Awerbuch algorithm [VA], the algorithms seem to defy comprehension by everybody but the designers, let alone the proofs of correctness. This undesirable state of affairs is worst in the case of multiwriter algorithms [VA], [PB], and [IL]. One reason is that the semantics of the partially ordered timestamp system used is difficult, and requires involved scaffolding constructions and correctness arguments. To demonstrate the simplicity of the solution in this abstract, we first concentrate on the complete construction of an atomic multiwriter variable from atomic multreader variables, together with a complete proof of correctness. This multiwriter construction achieves the best complexity among known such constructions: e.g., $O(n^2),3n$ versus $\Omega(n^2),\Omega(n^2)$ in [PB] (categories as in table). We then generalize the construction to do n-reader n-writer directly from 1-reader 1-writer.

Conventions for multiwriter from multireader solution. There are n processors in the system. Each processor i owns a multi-reader 1-writer shared atomic variable, named R_i. Only processor i can write R_i. Every processor can read all variables $R_1,...,R_n$, one at a time. We implement an atomic variable that can be read and written by all n processors under the following conditions. The readers and writers of the shared variable are assumed to be totally asynchronous. Each read and write must consist of an a priori bounded number of elementary actions, i.e., the program that the reader or writer executes is loop-free (e.g., no busy waiting).

Using unbounded tags, we can implement such a variable as follows. Each tag is a pair (t,i), where t is a natural number (a time stamp) and i is a processor index between 1 and n. The index i is the same as the register number in which t is written, so it needs not be written explicitly. Each register R_i contains initially value 0 and tag $(1,i)$. This protocol is used to derive the final one.

Protocol 1.

Processor i writes value:

1) Read all variables $R_1, \ldots, R_n$.

2) Select the lexicographical maximal tag, say (t,m).

3) Write value and $(t+1,i)$ to own variable R_i.

Processor i reads value:

1) Read all variables $R_1, \ldots, R_n$.

2) Select the lexicographical maximal tag, say (t,m).

3) Return value@(t,m).

Lemma 1. *Protocol 1 implements an atomic multiwriter variable.*

Proof. The correctness of this scheme follows immediately by the methods in [AKKV] or [Ly]. We have delegated the proof to the Appendix. ●

The core of our bounded protocol is simply the unbounded tag protocol above, with all timestamps reduced modulo $4n$. The only difficulty is how to choose the most recent timestamp. We will implement a simple mechanism to "kill" old timestamps, therefore keeping the eligible candidates of recent timestamps in a limited size window of size $2n$: Each write execution 'shoots' once at every operation execution it observes. (It shoots the finished ones as well as the ongoing ones.) To avoid 'mutual elimination', an operation execution gets 'killed' if it is shot twice by the same writer. If an ongoing operation execution notices it has been killed then it aborts, and, in case of a read, returns the value written by the killer. This is always safe, since the aborted operation execution completely overlaps the operation execution that killed it. In case it is a write, it can be ordered just before the killer; if it is a read then it can be ordered just after the killer. Each non-aborting operation execution considers only values of writes that have not been killed. At the time instant an operation execution a starts, all alive timestamps must be within a window of size n since the operation executions choose successive timestamps around the cycle and a timestamp will be killed in $\leq n$ writes (some writer wrote twice). During a, if any writer writes twice, then a gets killed. So if a is not killed, then the timestamps it sees are all in a window of size $2n$. The details are given below.

2. The Bounded Protocol for (1,n) → (n,n)

We transform Protocol 1 to the target algorithm (Protocol 2) and show that correctness is preserved. Protocol 2 uses $4n$ timestamps in cyclic order. In a run executed by Protocol 2, each write writes the same timestamp modulo $4n$ as the corresponding write in Protocol 1, and each read returns the value associated with the same (timestamp , index) pair, the timestamp modulo $4n$, as the corresponding read in Protocol 1. With each operation execution a (a read or a write) we associate a *frame(a)* of the following format:

operation a of processor i	
DIE§	*DIE¶*
SHOOT§	*SHOOT¶*
$timestamp^a$	
$Value^a$	

Figure: Frame of Protocol 2.

DIE§,*DIE¶*,*SHOOT§*,*SHOOT¶* are arrays of $2n$ bits each. The frame contains the current status of an operation execution: a timestamp, a value, and above arrays to record the recent shooting history. Each register R_i holds two frames. So at any fixed time, the system contains (not necessarily distinct) timestamps in $2n$ distinct frames. We index these frames by $1,2,\cdots,2n$. Register R_i holds frames $2i-1$ and $2i$. We denote the index of $frame(a)$ by $index_a$. The shooting is implemented by the DIE and $SHOOT$ arrays.[1)] These arrays have the following meaning:

1) The notion we use to implement actions like creation of frames or shooting, can be compared to a hotel switch. A hotel switch is a switch that can be switched from two different locations. I.e., there is a light switch upstairs and downstairs. From both one can switch the light on or off. The light (on or off) is a shared variable between two parties, one

$DIE_0^a(index_d) = SHOOT_0^d(index_a)$ and $DIE_1^a(index_d) = SHOOT_1^d(index_a)$ iff operation execution a is *killed* by the operation execution d, the *killer*. The key idea of the protocol is as follows: Mimic the unbounded protocol around the $4n$ cycle. The *only* difficulty is that the old timestamps maybe confused with the new ones when looping back. This is resolved as follows. Each time a writer finishes a write, it shoots the frame of every other operation once. So each out of date timestamp gets eliminated (killed) after n writes (shot twice by at least one writer).

Protocol 2.

We first implement 3 macro commands (k is a local binary variable of each processor):

(i) **Initialize frame**(a): R_i contains two frames: the most recent *frame*(b) with index $2i-k$, and the previous *frame*(c) with index $2i-\bar{k}$. To start an operation a, *frame*(c) is re-initialized as *frame*(a): Read all $SHOOT_{\bar{0}}$, $SHOOT_{\bar{1}}$ arrays of $R_1, \ldots, R_n$. Then, in one atomic write, processor i sets *frame*(a) with EMPTY timestamp (which is less than all other timestamps) field and EMPTY value field, $SHOOT_l^a := SHOOT_l^b$, and for any *frame*(d) read above $DIE_l^a(index_d) :\neq SHOOT_l^d(index_a)$, for all $l=0,1$.

(ii) a **shoots** b: The operation execution a *shoots* b by setting $SHOOT_k^a(index_b) := DIE_k^b(index_a)$.

(iii) a **aborts** means: operation execution a terminates without further changing any local or register variable.

An operation **dies** if it is shot twice by the same writer. In step 2.2) of read/write protocol below, a timestamp t is **maximal** in the set of alive timestamps, if the set does not contain $t+i$ mod $4n$, $1 \leq i \leq 2n$. **Initialize for run:** $k=0$, and DIE, $SHOOT$ arrays such that nobody shoots anybody. Timestamps and values in frames in all R_i are initialized as in Protocol 1.

Processor i writes value:

```
0)   Initialize frame(a). /* frame index $2i-\bar{k}$ */
1)   Read all variables $R_1, \ldots, R_n$, to select maximal timestamp in step 2.2).
2.1) Read all variables $R_1, \ldots, R_n$. If shot twice by one writer, then abort, else
2.2) Select lexicographical maximal tag, say $t$, using alive frames from step 1).
3)   In frame(a), write value and $[t+1 (mod\ 4n), i]$ to the value and timestamp fields, and shoot
     every other write/read frame read in step 1) once using $SHOOT_k^a$ (all in 1 atomic write to
     $R_i$); $k := k+1 (mod\ 2)$.
```

Processor i reads value:

```
0)   Initialize frame(a). /* frame index $2i-\bar{k}$ */
1)   Read all variables $R_1, \ldots, R_n$, to select maximal timestamp in step 2.2).
2.1) Read all variables $R_1, \ldots, R_n$. If shot twice by one writer, then return the killer's value and
     abort; else
2.2) Select the lexicographical maximal tag, say $(t, m)$, using alive frames from step 1).
3)   Return value@$(t, m)$.
```

Lemma 2. *Protocol 2 implements an atomic multiwriter variable.*

Proof. Consider a run of Protocol 2 with aborted operations deleted. Those aborted

at each switch, that can be set and reset by both parties using their own switches (i.e., local variables).

operations can be inserted at will after we serialize other operations since no read returns the value of an aborted write and

Claim 1. In Protocol 2, if an operation execution aborts, then it overlaps completely a write by the killer. Therefore the aborted read/write can be inserted after/before the killer write in the atomic order.

Proof of Claim 1. If the frame of operation a of processor W' is shot twice by writer W, then the killer bits in W's register are both set after the operation execution a scanned those bits in step 0). Since it takes W two writes to set both killer bits, the second write is completely overlapped by a (also the most recent write by W). •

Now let the same run (without the aborted actions) be executed by the unbounded Protocol 1 such that the corresponding subactions happen at same time. By Lemma 1, Protocol 1 induces a total order $a_1, a_2, \cdots$ of the set of operation executions according to the atomicity requirement. Let w_i be the write associated with the tag selected by a_i in step 2) of Protocol 1. The lemma follows from the following claim,

Claim 2. In Protocol 2, for all i, a_i also selects the tag associated with w_i in step 2.2).

Proof of Claim 2. Induction on a_i's. Assume that the corresponding atomic subactions in both protocols take place in precisely the same instants. (I.e., the atomic subactions in step 1), for both read's and write's, and those in step 3) of the write's.)

Base. Both protocols are identically initialized.

Induction. Assume that the claim is true for $i=1, \cdots, k-1$. Consider action a_k (which did not abort) in Protocol 2. Let S_i= set of writes whose timestamps were obtained by a_k in step 1) of Protocol i, for $i=1,2$. Since the scans in step 1) of the two protocols were performed at the same times, by induction assumption and Protocol 1, a_j is not in S_1, and therefore not in S_1, for any $j>k$, and moreover $S_1 \subseteq S_2$.

Let step 0) finish at time t_0, and step 2.1) start at time t_2. At time t_0, We claim that all alive operations are within an interval of size n. This is because, at step 3) of the writer algorithm of Protocol 2, every writer shoots every other writer's frames every time it finishes a write. (Claim 3 guarantees once a timestamp is dead, it does not become alive.) From t_0 to t_2, at most $<n$ consecutive timestamps can be written. If not,then some writer would shoot a_k's frame twice and keep the killing bits until $frame(a_k)$ is replaced; then a_k would detect this after t_2 in its scan in step 2.1 and abort, a contradiction. Therefore the alive timestamps observed by a_k are clustered in a window of size $2n$ which include the largest timestamps seen by the unbounded protocol, which corresponds to the maximal timestamps among the alive timestamps seen by a_k. So the bounded and unbounded protocols choose the maximal tag corresponds to a same write by the same selection rule, provided

Claim 3. a_k can distinguish alive and dead timestamps. Dead timestamps stay dead.

Proof of Claim 3. Assume operation a, performed by writer W_a, killed b. By Protocol 2, b never changes its *DIE* arrays once initialized. W_a will keep the killing bits for b until it sees that $frame(b)$ is replaced by a fresh frame. After $frame(b)$ is removed, it takes two writes of W_a to change all the killing bits for b because W_a keeps two frames and it replaces the older frame first. Now in step 1) of Protocol 2, if a_k sees both $frame(b)$ and the killing bits from W_a, or a_k does not even see $frame(b)$, then a_k sees b is dead. If a_k only sees $frame(b)$ but did not see the killing bits from any of the two frames of W_a, then W_a wrote twice after $frame(b)$ is removed and hence after a_k sees $frame(b)$. Hence W_a shoots a_k twice in the meantime and a_k must detect this and abort at step 2.1) of Protocol 2, contradiction. ●

3. Direct Construction of (n,n)-Registers from (1,1)-Registers

Instead of Protocol 1, we start with the Vitanyi-Awerbuch unbounded tag algorithm [VA]. Correctness proofs can be found in [VA], [Ly], and [AKKV]. We assume familiarity with this algorithm. The killing strategy is so robust, that the transformation to bounded tag algorithm is completely analogous to that above, as is the correctness proof. The added indeterminacy of the final write in each operation execution only requires to make the cycle larger. So we only need to modify as follows.

- Each 1-reader 1-writer variable contains three frames. In the frames, the index number of the writer is written explicitly. When a read does not abort, then it replaces in its two most recent frames only the values, timestamps and processor indexes by those of the two most recent frames in the selected subregister. (Not the *SHOOT* and *DIE* arrays.) This, by a final write in step 3), just like in a write execution. However, the read does not shoot.
- Both read and write do the initialization write in step 0), and the final write in step 3) by atomic writes to all subregisters in its row.
- An operation execution must be shot three times to be killed. Hence the frames have $3n$-bit arrays DIE_i and $SHOOT_i$, $i=0,1,2$. If an operation is killed, then we count the write before the last as the killer. (This write is neatly sandwiched between the first shooter and the last, and thus is completely overlapped, even though the final write action of [VA] involves writing to $n-1$ variables shared with the other processors.) We can determine the frame of the one but last write, because it is shot once by the own writer.
- When a read aborts, it returns the value of the killer, and then just quits,

The changes imply that the window in which viable candidates cluster is less than cn, so $2cn$ timestamps suffice, for some small constant c. Each register holds 3 frames, with altogether $O(n)$ bits. The protocol still uses 3 scans of n column variables, but now also 2 writes to n row variables, i.e., $5n$ total.

4. Appendix

Proofsketch Lemma 1. Order the set of actions A such that action a precedes action b, $a \rightarrow b$, if $f(a)<s(b)$. (Here $s(a)$ is the start time, and $f(a)$ is the finish time of an action a.) Define the reading mapping π, as a mapping from reads to writes by: if r is a read that returns the value of a write w, then $\pi(r)=w$. We define the pair (A,π) to be *atomic* if we can extend $\rightarrow$ to a total order $\rightarrow'$ such that (i) $\pi(r)\rightarrow' r$, and (ii) there is no write w such that $\pi(r)\rightarrow' w \rightarrow' r$. A necessary and sufficient condition for atomicity is given as follows. π factors the set of actions A in equivalence classes $[w]=\{x : x=w \text{ or } \pi(x)=w\}$. We define a relation $\rightarrow_\pi$ on the set of equivalence classes by $[w]\rightarrow_\pi[w']$ if there are $a\in[w]$ and $b\in[w']$ such that $a\rightarrow b$. It is not difficult to see [AKKV] that (A,π) is atomic iff (i) not$(r\rightarrow\pi(r))$, and (ii) $\rightarrow_\pi$ is acyclic. Now, (i) holds trivially. So we only need to show (ii). Define $tag(x)$ equal (t,i) if x is a write by i that writes timestamp (t,i), or if x is a read that returns value@(t,i). If $[w]\rightarrow_\pi[w']$ then there is an action x in $[w]$ and an action y in $[w']$, such that $x\rightarrow y$. Since the value of w is not readable before its very end, $f(w)\le f(x)$, and therefore $w\rightarrow y$. Let $<_{lex}$ denote the lexicographic order on pairs of integers. If $y=w'$ then the protocol sets $tag(w)<_{lex}tag(w')$. If y is a read, and $tag(w)>_{lex}tag(w')$, then y chooses w, a contradiction. Therefore, $[w]\rightarrow_\pi[w']$ implies $tag(w)<_{lex}tag(w')$. Since the lexicographic order is total, $\rightarrow_\pi$ has no cycles.●

References

[AKK] B. Awerbuch, L.M. Kirousis, E. Kranakis, P.M.Vitányi, "On Proving Register Atomicity", CWI Tech Rept, Amsterdam, May 1987.

[Bl] B. Bloom, "Constructing Two-writer Atomic Registers," PODC-87.

[BP] J.E. Burns and G.L. Peterson, "Constructing Multi-reader Atomic Values From Non-atomic Values", PODC-87.

[IL] A. Israeli and M. Li, "Bounded Time-Stamps", FOCS-87.

[KKV] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, "Atomic Multireader Register", Information and Computation, to appear. (Also, Proc. 2nd International Workshop on Distributed Computing, Amsterdam, July 1987.)

[La] L. Lamport, "On Interprocess Communication Parts I and II", Distributed Computing, Vol. 1, 1986, pp. 77-101.

[Ly] N. Lynch, "MIT 6.852 Class Notes", Oct. 3, 1986.

[NW] R. Newman-Wolfe, "A Protocol for Wait-free, Atomic, Multi-Reader Shared Variables", PODC-87.

[PB] G.L. Peterson and J.E. Burns, "Concurrent reading while writing II: the multiwriter case", FOCS-87

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

[SAG] A.K. Singh, J.H. Anderson, M.G. Gouda, "The Elusive Atomic Register Revisited", PODC-87.

[VA] P.M.B. Vitányi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware", FOCS-86. (Errata, FOCS-87)