Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Computation 200 (2005) 62–106
Information and Computation
www.elsevier.com/locate/ic

Time and space optimal implementations of atomic multi-writer register☆

Amos Israeli [a,*], Amnon Shaham [b]

[a] *School for Computer Science and Mathematics, Netanya Academic College, Israel*
[b] *Intel, Israel*

Received 20 September 2000; revised 3 July 2004
Available online 23 May 2005

Abstract

This paper addresses the wide gap in space complexity of atomic, multi-writer, multi-reader register implementations. While the space complexity of all previous implementations is linear, the lower bounds are logarithmic. We present three implementations which close this gap: the first implementation is sequential and its role is to present the idea and data structures used in the second and third implementations. The second and third implementations are both concurrent, the second uses multi-reader physical registers while the third uses single-reader physical registers. Both the second and third implementations are optimal with respect to the two most important complexity criteria: their space complexity is *logarithmic* and their time complexity is *linear*.

1. Introduction

1.1. Problem description

At the most basic level of asynchronous interprocessor communication, data are transferred via *shared memory*. A *register* is a very simple model for shared memory-based communication.

☆ Partially supported by NWO through NFI Project ALADDIN under Contract No. NF 62-376. A preliminary version of this paper was presented in the *11th Annual ACM Symposium on Principles of Distributed Computing*, August 1992, Vancouver, Canada.

* Corresponding author. Fax: +972 9 860 7825.
E-mail addresses: amos@netanya.ac.il (A. Israeli), amnon.shaham@intel.com (A. Shaham).

0890-5401/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2004.11.004

Processors access registers by executing *read* and *write operations*. Each register has a set of *writer* processors, a set of *reader* processors and a set of *permitted values*, including a distinguished value called the register's *initial value*. A register is *atomic* if each operation is executed instantaneously and each read operation returns the value written by the most recent, preceding, write operation, or the initial value if no such preceding write operation occurred. Each atomic register is further classified by the number of its writers, the number of its readers and the number of its permitted values. Once atomic registers are defined and classified, it is natural to compare the relative computational power of various kinds of registers. In this paper, we study the computational resources needed to implement a multi-writer atomic register using single-writer atomic registers.

Informally, an *implementation* of a *logical* register using a set of *physical* registers consists of a hardware arrangement of the physical registers and two programs that are called the *writer protocol* and the *reader protocol*. Both protocols are composed of operations of the physical registers (which are called *physical* operations) and constitute the operations of the logical register (which are called the *logical* operations). The set of processors is partitioned into logical writers and logical readers, that is, writers and readers of the logical register. For simplicity we assume that each processor is either a logical writer or a logical reader though in reality a processor can function as both.

Whenever a processor "wishes" to execute a logical operation, it starts executing its protocol. The execution of a physical (logical) operation is called a physical (logical) action. The physical actions executed by a processor during a logical action may be interleaved with physical actions of other processors and since the system is entirely asynchronous there is no bound on the number of physical actions (of other processors) between any two consecutive physical actions of any processor. In particular, a processor may *crash*, that is stop execution forever, in the middle of any logical action.

Informally, an execution of the system is a sequence of physical actions partitioned into logical actions. The correctness condition we use is called *linearizability*. A *linearization* of an execution of some register implementation is an assignment of a distinct *linearization point* for each logical action such that the induced sequence of logical actions preserves the order of non-overlapping actions and each read operation returns the value written by the most recent, preceding, write operation, or the initial value if no such write operation occurred. An implementation is *linearizable* if all its executions are linearizable. To ensure resiliency for crash faults and to avoid the use of mutual-exclusion techniques that eliminate concurrency, it is required that the reader and writer protocols are *wait-free*, that is, the number of physical actions a processor executes during a single logical action is bounded from above, where the bound may depend on the number of processors in the system.

1.2. Complexity measures

Throughout this paper w and r are used for the number of writers and readers, respectively, and $n = w + r$ is the total number of processors in the system. A register with w writers and r readers is denoted as a (w, r)-register. In *label-based* implementations, the physical registers are divided into two fields: a *value* field and a *label* field. The value field holds a permitted value, or a finite set of permitted values. The only operations which access the value field are copying a value to this field or from it. The label field holds all the coordination information needed for the implementation.

We deal with label-based implementations of two types, according to the physical registers used: In type $(1,n)$, each logical writer owns[1] an atomic $(1,n)$-register,[2] via which it communicates with all other processors. In type $(1,1)$, each processor communicates with every other processor via a $(1,1)$ atomic register. It should be noted that if two registers of the same owner have the same set of readers, they can be joined to a single register in which the two registers are represented as fields; thus we assume that the sets of readers of every two registers of the same owner are distinct. Under this assumption, the complexity of label-based implementations is measured by:

1. *Space complexity*. The maximal size of a label field of any physical register. (This criterion is often called *label-size*.)
2. *Time complexity*. The maximal number of physical actions executed during a single logical read or write operation.

Note. The definition of space complexity enables us to ignore the size of the value set of the implemented register. In case the value field holds more than a single value, the number of values in the value field should also be considered and added to the space complexity. In the second and third implementation presented in this paper, the value field has two and four values, respectively. We regard 4 as a small enough constant and conveniently ignore it.

A third and much less common complexity measure is parallel-time complexity. In order to analyze the parallel-time complexity of an implementation it is assumed that each processor P can write in (respectively, read from) k of its n registers ($k \leqslant n$) in parallel, using a single operation. Furthermore, it is assumed that whenever p issues a parallel write operation, all k actions are executed independently and any of these actions interleaving among themselves as well as with actions of other processors is possible. The only restriction is that P may not continue its execution until all k actions are completed. Under these assumptions we define:

3. *Parallel-time complexity*. The maximal number of physical actions, including *Parallel Write* and *Parallel Read*, executed during a single logical read or write operation.

1.3. Results of this paper

We present three bounded, label-based implementations for an atomic, multi-writer, multi-reader register, with *logarithmic* space complexity. The first implementation is sequential and its role is to present the ideas and data structures used by the second and third implementations. The second and third implementations are the first concurrent implementations of multi-writer register with *logarithmic* space complexity.

The second implementation is of type $(1,n)$; in this implementation communication is one sided: Writers communicate to readers (and to other writers) while readers do not write. Each writer

[1] The writer of every physical register is called its *owner*.

[2] Throughout this paper we assume that each processor communicates with itself via a register. This assumption is not essential and is used to obtain simpler code and cleaner expressions, e.g., $(1,n)$-register instead of $(1,n-1)$-register.

owns an atomic $(1,n)$-register which can be read by all processors, writers and readers. The space complexity of this implementation is $\Theta(\log w)$; by the lower bounds of [3,19], this bound is optimal for *label-based implementations*. For general implementations, it is not hard to prove that the number of values in each of the registers used in any implementation is bounded below by the size of the value set of the implemented register. Hence, if the number of permitted values of the implemented register is polynomial in n, the register size is $\Omega(\log n)$. Therefore, the space complexity of this implementation is optimal for any implementation of an atomic register whose value set size is polynomial in n. The time complexity of this implementation is $\Theta(w)$ which is optimal for any implementation. The parallel-time complexity of the $(1,n)$ implementation is $\Theta(w)$.

The third implementation is of type $(1,1)$, in this implementation communication is two sided: Each processor, writer or reader, communicates with each other processor via an atomic $(1,1)$-register. In [9], it was proved that two-sided communication is necessary for implementations of type $(1,1)$. The space complexity of this implementation is $\Theta(\log n)$ and the time complexity is $\Theta(n)$. In case $w = \Theta(n)$, the space complexity is optimal for label-based implementations. For general implementations, if the size of the value set is polynomial in n then the space complexity is optimal. The time complexity is optimal for any implementation. The parallel-time complexity of the $(1,1)$ implementation is $\Theta(n)$.

One may wonder which of the two concurrent implementations is "better"? The answer depends on the type of physical registers at our disposal. If the available physical registers are of type $(1,n)$ then they can be used as $(1,1)$ in the $(1,1)$ implementation, but in this case the space and time complexity are $\Theta(\log n)$ and $\Theta(n)$, respectively, instead of $\Theta(\log w)$ and $\Theta(w)$, and the number of needed registers is n^2 instead of n. Thus, in this case the answer is clear.

If the available physical registers are of type $(1,1)$ then the $(1,n)$ implementation can be used if the required $(1,n)$-registers are implemented from $(1,1)$-registers. The best implementation for a $(1,n)$-register from $(1,1)$-registers is obtained from the multi-writer implementation of [13]. Adapting this implementation for a single writer yields an implementation of $(1,n)$-register in which the single writer owns $(1,1)$-registers whose label-size is $\Theta(n)$ while the label-size of the readers' registers is $\Theta(1)$. Thus, the space complexity is $\Theta(n)$ and its time complexity is $\Theta(n)$ as well. In order to implement the w $(1,n)$-registers, each writer of the implemented (w,r)-register uses n $(1,1)$-registers whose label-size is $n + w - 1$ bits. The space complexity of the combined implementation is $\Theta(n + w \log w)$. The time complexity for reading or writing a single value is $\Theta(n)$, thus, the time complexity of the combined implementation is $\Theta(w \cdot n)$. Obviously, when the available physical registers are $(1,1)$, the $(1,1)$ implementation whose space and time complexity are $\Theta(\log n)$ and $\Theta(n)$, respectively, is superior to the $(1,n)$.

1.4. Previous work

Peterson, in [16], presented the first implementation of a register by another register, Misra, in [15], gave axioms for shared memory systems and Lamport [10,11] was the first to formalize the notion of a register implementation. In [10,11], Lamport showed that an atomic $(1,1)$-register with any value set can be implemented using very weak registers, namely binary, safe, $(1,1)$-registers. Several papers, motivated by the work of Lamport, studied the intriguing problem of implementing atomic, multi-writer, multi-reader registers. The simplest such implementation was presented by

Table 1
Implementations of type $(1, n)$

	Refs. [17,18]	Ref. [7]	Ref. [4]	This paper	Ref. [2]
Space	$\Theta(w)$	$\Theta(n)$	$\Theta(w)$	$\Theta(\log w)$	$\Theta(\log w)$
Time	$\Theta(w^2)$	$\Theta(n)$	$\Theta(w^2 \log w)$	$\Theta(w)$	$\Theta(w)$

Table 2
Implementations of type $(1, 1)$

	Ref. [20]	Ref. [13]	This paper
Space	unbounded	$\Theta(n)$	$\Theta(\log n)$
Time	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Parallel time	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$

Vitányi and Awerbuch in [20]. They present a label-based implementation of an atomic (w, r)-register, using atomic $(1, 1)$-physical registers. In the implementation of [20], the labels are unbounded counters used as time-stamps, hence this implementation has *unbounded* space complexity. The actual size of a label in any logical action is logarithmic in the number of write actions performed prior to that action. The time complexity of this implementation is linear in n, the total number of processors.

Some researchers have devised bounded label-based implementations of type $(1, n)$, for atomic multi-writer, multi-reader registers: The first implementation was proposed in [20] and was found to be erroneous. The second implementation was presented by Peterson and Burns in [17]—that implementation has a bug which was discovered and corrected by Schaffer in [18]. Its space complexity is $\Theta(w)$ and its time complexity is $\Theta(w^2)$. Israeli and Li, in [7], suggested *bounded time-stamps* as a bounded primitive to capture the precedence relationship among asynchronous processors. Using bounded time-stamps they presented an implementation whose space and time complexity are $\Theta(n)$. The correctness of the [7] implementation was never fully proved. Another implementation with higher complexity was presented by Dolev and Shavit [4]. Abraham in a manuscript dated 1991 presents an implementation whose space and time complexity are $\Theta(w)$. A later version of this paper with stronger results that are influenced by the results of this paper appears in [2]. The various $(1, n)$ implementations are compared in Table 1. The parallel-time complexity of none of these $(1, n)$ implementations was analyzed.

The only two implementations of type $(1, 1)$ are the original unbounded implementation of [20] whose time complexity is $\Theta(n)$ and the implementation of Li, Tromp and Vitányi, presented in [13]. The space and time complexity of the [13] implementation is $\Theta(n)$, and its parallel-time complexity is a small constant. The various $(1, 1)$ implementations are compared in Table 2.

1.5. Discussion

Prior to this work, all proposed bounded concurrent implementations have a space complexity which is *linear* in the number of writers, and in some cases even in the total number of processors. In [14], Li and Vitányi presented a *sequential* implementation (sequential implementations are correct only for sequential executions in which the *logical* actions are executed sequentially,

without overlapping) with $\log w$ space complexity. The sequential implementation of Li and Vitányi uses the *id*s of the processes, or in other words, is not anonymous. For an anonymous implementation their method requires labels of size $2 \log w$, since the processor's *id*s are added to the labels.

Later, it was proven by Cori and Sopena in [3] that an anonymous implementation for w writers should have at least $2w - 1$ distinct labels. They also devised a sequential implementation with exactly $2w - 1$ labels which improved the space complexity of the sequential [14] implementation by a constant. In [19], it was proven by Tromp that the sum of sizes of label fields in non-anonymous implementations is at least $w \log w$ which translates to $\Omega(\log w)$ label size assuming that the size of all label fields is the same.

These results leave an *exponential* gap between the lower and upper bounds on the space complexity of label-based implementations of atomic registers. In a way, this bound represents the cost of *concurrency*: the lower bounds of [3] and [19] consider only the combinatorial requirements for identifying the last written value. For this reason, these lower bounds hold for sequential and concurrent implementations. The extra complexity of concurrent implementations seems to be incurred by the need to deal with concurrency aspects. All concurrent implementations before this paper use direct binary comparison between labels of every pair of processors to determine their precedence relations and therefore they all require (at least) *linear* space.

The significance of this gap is further emphasized when one considers a related problem, namely: an implementation of an atomic register which is correct for executions whose length is *polynomial* in the number of processors. One may motivate this definition by arguing that in real life, the probability for longer executions is so low that the cost of allowing them must be taken into account. For polynomially long executions, the space complexity of the [20] implementation is *logarithmic* while the space complexity of all previous bounded protocols is *linear*. In other words, for a problem that might be considered more practical, the protocol of [20] supersedes all other implementations by an *exponential* factor. The results of this work show that boundedness can be achieved with no more than *logarithmic space complexity*, thus closing the exponential gap discussed above.

The problem of implementing a multi-writer atomic register is well known. As we pointed out, several erroneous or incomplete solutions have been published and debugging them was complex and controversial. Given the history of the problem we believe that no implementation is worth anything without a complete detailed correctness proof. In this work, we fulfill this obligation and present full proofs, without any shortcuts, to both implementations. Unfortunately, the proofs are often not intuitive, very technical and uninspiring. We regard the presentation of a formal verification system for register implementations as an important open problem.

1.6. Paper organization

The rest of this paper is organized as follows: In Section 2, we formally define the model of computation and the implementation problem. In Section 3, we explain the data structure used by our protocols and present a sequential implementation. The sequential implementation serves as an exposition for the ideas which are later used in the concurrent implementations. The $(1, n)$ and the $(1, 1)$ implementations are presented in Sections 4 and 5, respectively. Concluding remarks are presented in Section 6, the correctness proof of the hand-shake mechanism appears in Appendix A.

2. Model and requirements

In this section, we define the model of computation and the atomic register implementation problem. A system consists of a set of processing entities called *processors*, and a set of memory entities called *atomic registers* (for brevity we use the term *registers* throughout this paper). Each register has a set of *writer* processors, a set of *reader* processors and a set of *permitted values*, including a distinguished value called the register's *initial value*. Processors access registers by executing *read* and *write operations*. A *write* operation to register *REG* is executed by some writer of *REG*, the operation gets a permitted value of *REG* as an input parameter and it stores the value in *REG*. Analogously, a *read* operation from *REG* is executed by some reader of *REG*, the operation retrieves the (permitted) value stored in *REG* and the value is returned as an output parameter.

Register *REG* is *atomic* if each operation is executed instantaneously and each read operation returns the value written by the most recent preceding write operation, or *REG*'s initial value if no such write operation exists. A *processor* is a finite state machine. For convenience, we describe a processor by its *protocol* whose building blocks are *instructions* where each instruction corresponds to a single state-transition. Each instruction starts with an internal computation which is succeeded by at most one operation. Each protocol has a distinguished instruction corresponding to the state machine's initial state, that is called the protocol's *initial instruction*.

System executions are described under the interleaving model: a system's global state is described by its *configuration*—a vector containing the state of each processor and the value of each register. The system's *initial configuration* contains the processors' initial states and the registers' initial values. The execution of an instruction is called an *action*. We assume that each processor repeatedly executes its protocol and that the actions of the system's processors are interleaved. A system execution is a sequence of configurations and actions $E = c_0, d_1, c_1, \ldots, c_{i-1}, d_i, c_i, \ldots$, where c_0 is the system's initial configuration and for every $i > 0$, c_i is obtained from c_{i-1} by action d_i. Configuration c_i is called the *result configuration* of action d_i. Whenever convenient we omit the d_i's and describe an execution by the sequence $c_0, c_1, \ldots$

Now, we define the implementation problem for atomic registers: an atomic register is a *concurrent object*, see [5]. A concurrent object can be specified either as an automaton, see [12], or by a set of axioms, see [15], or by a set of sequential executions, see [6]. Regardless of the method of specifying the concurrent object, most works assume the interleaving model and describe executions in the way we presented above. Intuitively, a system is an implementation of logical register *REG*, if the system processors can be divided into writers and readers and a single execution of a writer (reader) protocol *can be looked at* as a write (read) action to *REG*. If the physical register(s) are equal to the logical register, the problem is trivial, thus, to make the problem meaningful, the implemented register should have either a larger number of readers, or a larger number of writers, or a larger set of values.

Instead of giving a formal definition of an implementation, we refer the reader to [5,l] in which the notion of implementation is discussed and formally defined. Here, we resort to an informal description which explains what we do without any ambiguities: An *implementation* of a *logical register* is a system whose registers are called *physical registers*. The operations that access the physical registers are called *physical operations*. A *writer* (*reader*) is defined by the *writer protocol* (*reader protocol*), whose building blocks are physical operations. The writer protocol has one input parameter which is the value that should be stored in the logical register. The reader protocol should be

exited through instruction *return* that does not contain any physical operation but rather returns a value which is the result of the logical read action.

Consider an *execution* of an implementation in which each processor executes its protocol repeatedly and the actions of the processors are interleaved. Each execution of the writer (reader) protocol constitutes a *logical write* (*logical read*) action. Let a be a logical action of processor P_i. The actions of P_i, executed during a are the *physical actions* of a. Let $d_{i_1}, c_{i_1}, d_{i_2}, \ldots, d_{i_t}, c_{i_t}$ be the physical actions and resulting configurations of logical action a. Configuration c_{i_1-1} is called a's *initial* configuration, configuration c_{i_t} is called a's *final* configuration and the interval $[c_{i_1-1}, \ldots, c_{i_t}]$ is called a's *execution interval*. It should be noted that interval $[c_{i_1-1}, \ldots, c_{i_t}]$ may contain actions of all processors (and not necessarily of P_i alone). We prove the correctness of the implementations under the *linearizability* correctness condition, [6]. An execution is *linearizable* if each logical action can be assigned a *linearization point*, such that the sequence of linearization points preserves the order of non-overlapping logical actions and each logical read action r returns the value written by w, where w is the action that is linearized last among all logical write actions linearized before r. In all linearizations we present, every linearization point lies within the execution interval of its logical action, which trivially preserves the order of non-overlapping actions.

An implementation is *linearizable* if all its executions are linearizable. An execution in which the physical actions of each logical action are executed one after the other, with no interleaving with the physical actions of any other logical actions, is called *sequential*. An implementation is *sequential* if all its sequential executions are linearizable. A protocol is *wait-free* if all its executions consist of a bounded number of physical actions, where the bound may depend on the number of processors in the system. We require that the logical operations are wait-free.

3. Basic principles

In this section, the precedence graph method, used in all three implementations, is reviewed. Then, the actual precedence graph on which the implementations are based is presented. The section is concluded by presenting a sequential implementation of a multi-writer, multi-reader register from single-writer, multi-reader registers. The sequential implementation is used to demonstrate the main features shared by all three implementations.

3.1. Time-stamps and the precedence graph method

Many concurrent protocols use the *Natural Time-Stamps Scheme* to represent temporal precedence relations among protocol actions: Each action of the protocol is labeled with some natural number called the action's *time-stamp* and the time-stamp of an action is always *larger* than the time-stamp of every preceding action. In [7], Israeli and Li observed that the natural time-stamp scheme can be looked at as a graph, where each time-stamp is a node and the node of every time-stamp *dominates* the nodes of all *preceding* time-stamps. Following that observation, they proposed to keep the basic idea in which nodes of earlier actions are dominated by nodes of later actions but to replace the infinite graph of the natural time-stamps scheme with some finite graph.

The first problem to overcome when using a finite graph to represent temporal precedence relations is the fact that *the number of time-stamps is finite*. Let us elaborate on this problem: consider

the situation when a protocol needs to pick a time-stamp for some new action during some execution. When the protocol uses the natural time-stamp scheme, it simply picks a number larger than any number used earlier in that execution. Thus, in this case, every logical action is stamped with a *unique* time-stamp, all time-stamps are ordered by the temporal precedence relation and no confusion can occur. Now, consider the same situation, when the time-stamp set is finite: for long enough executions, the time-stamp collection is eventually depleted and old time-stamps must be reused. Call a time-stamp *alive* if it exists in the memory of some processor. Whenever a time-stamp is picked for some new action, it is obvious that the protocol should not pick an alive time-stamp, but this is not enough: The protocol must find all alive time-stamps and pick a time-stamp that dominates all of them. In this way, the set of all alive time-stamps is always ordered by temporal precedence relation. This problem was posed and solved in [7]. For concurrent time-stamp schemes, the problem was solved by several protocols, see, e.g. [4].

By definition, each concurrent time-stamp scheme enables determination of the precedence relation between every pair of alive time-stamps by a direct binary comparison. Therefore, each concurrent time-stamp scheme can be used to implement an atomic multi-writer register. As was proved in [7], the space complexity of any such time-stamp scheme is at least linear. In this work we use the precedence graph method, but in order to achieve *logarithmic* space complexity we must give up binary precedence comparability. Instead we resort to a new method of using precedence graphs. In this new method, precedence relations are represented using *directed paths*. All nodes on each directed path are temporally ordered while the temporal order among nodes on different paths is not determinable. Though our new approach cannot be used to find a complete temporal order among all alive time-stamps, it is sufficient to enable each read action to determine the most recent preceding write action, which is exactly the requirement for implementing an atomic register. A second difference of a technical nature is the reversing of domination order, namely: In the new method, nodes corresponding to *earlier* actions *dominate* nodes corresponding to *later* actions.

3.2. The precedence graph and its sub-trees

In this section we present the precedence graph used by the three implementations and the way in which it represents temporal precedence order:

Definition 1. Let $\mathcal{ID}$ be the set of *processor ids*, $\{0,1,2,\ldots,p\}$, and let $\mathcal{AD}$ be the set of *addresses* $\{0,1,\ldots,max\_add\}$. The number of *id*s, $p+1$, is equal to the number of processors in the system plus 1. The number of addresses, max_add+1, which determines the exact size of the precedence graph, is a function of the number of processors and it differs from one implementation to another. The precedence graph $\mathcal{P}=(V,E)$ is defined as follows:

- Each node is a quadruple of natural numbers. The set of nodes is a subset of $\mathcal{ID}\times\mathcal{AD}\times\mathcal{ID}\times\mathcal{AD}\cup\{v_0^0\}$, where $v_0^0=(0,0,0,0)$ is called the *root node*. The four components of node v are denoted by $v.id$, $v.address$, $v.tail\_id$ and $v.tail\_address$. A pictorial description of a single node appears in Fig. 1.
- The edges of $\mathcal{P}$ are encoded by the names of the nodes: Let u and v be two nodes. If $u.id=v.tail\_id$ and $u.address=v.tail\_address$ then there is an edge emanating from u and incoming to v. This

Fig. 1. A single node.

edge is denoted by (u, v) and u is called the *tail* node of v. To avoid non-trivial cycles we require that for every node v, $v.id \geqslant v.tail\_id$.
Note. The indegree of all nodes is 1. For convenience we ignore the self-loop that emanates from v_0^0 and regard its indegree as 0. For any other node v, $v \neq v_0^0$, the edge incoming to v is called *the edge of* v. In most cases it holds that for every node v, $v.id > v.tail\_id$. Self-loops are allowed only in special cases that will be described later.

In all three implementations, the implemented register is a (w, r)-register where w and r are used for the number of writers and readers, respectively, and $n = w + r$ is the total number of processors in the system. The writers of the implemented logical register are denoted by $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_w$ and the readers are denoted by $\mathcal{R}_{w+1}, \ldots, \mathcal{R}_{w+r}$. Execution number a of the writer protocol by $\mathcal{W}_i$ is denoted by L_i^a; i and a are called the *id* and the index of L_i^a, respectively. Logical action L_i^a computes a node, denoted by v_i^a, that is written into the *current* field of REG_i at the end of L_i^a. Node v_i^a remains in (the *current* field of) REG_i until the end of L_i^{a+1}, when v_i^{a+1} is written into (the *current* field of) REG_i. During this interval node v_i^a is called the *current node* of $\mathcal{W}_i$. Given an execution E, the current node of each logical action is completely determinable. Note, however, that the processors cannot compute the indices of the actions that generated the processors' current nodes.

For every current node of $\mathcal{W}_i$, v_i^a, it holds that $v_i^a.id = i$ and all these nodes are stored in REG_i. Therefore, the *id*, i, is omitted from the explicit encoding of v_i^a in REG_i, and v_i^a is specified by its *address*, *tail_id* and *tail_address* components. The logical value, written in L_i^a, is attached to v_i^a, but the protocols never use the logical value and it is omitted from the protocol's description.

All three implementations use a procedure called *collect* whose code appears in Fig. 2. Procedure *collect* computes and returns a subgraph of $\mathcal{P}$, called G, induced by the *current* nodes of all processors. In addition, *collect* computes and returns the set AD containing the *tail_address*-s of all collected nodes. Let us describe *collect*: the set V is initialized with *root node* v_0^0. Then nodes are *collected* by reading the processors' registers in ascending order of the processors' *id*s. After all

```
Procedure collect(G, AD)
        V := {v_0^0}
        AD := φ
        for j := 1 to w do
            v_j := read(REG_j.current)
            V := V ∪ v_j
            AD := AD ∪ v_j.tail_address
        end for
        G := IND(V)
        return(G, AD)
```

Fig. 2. The code for procedure *collect*.

nodes are collected, the procedure computes the subgraph induced by all nodes in V. Recall that the indegree of every node of $\mathcal{P}$, except the root node, is 1. Since G is a subgraph of $\mathcal{P}$, the indegree of each node in G is at most 1 and since there are no non-trivial cycles, every collected graph is a forest of rooted trees. One of these trees is always rooted at the root node while the other trees (if there are any) are rooted at some other nodes. Since we require that for every node v_i^a, $v_i^a.id \geqslant v_i^a.tail\_id$, the nodes on each directed path of the precedence graphs are ordered in an ascending order of their *id*s. From now on we do not use the fixed precedence graph explicitly, instead we use the term *precedence graphs* to refer to the subgraphs of $\mathcal{P}$, computed by *collect*.

The following example demonstrates the way in which the current graph is constructed: assume that the following set of values was read from $REG_1, \ldots, REG_6$:

$$REG_1 = (1,0,0), \quad REG_2 = (6,0,0), \quad REG_3 = (8,1,3),$$
$$REG_4 = (7,2,6), \quad REG_5 = (0,3,8), \quad REG_6 = (1,4,7).$$

In order to get the actual nodes that were collected, the *ID* of each processor should be appended as the first field of each node, thus the set V of collected nodes is:

$$v_0^0 = (0,0,0,0),$$
$$v_1^a = (1,1,0,0), \quad v_2^b = (2,6,0,0), \quad v_3^c = (3,8,1,3),$$
$$v_4^d = (4,7,2,6), \quad v_5^e = (5,0,3,8), \quad v_6^f = (6,1,4,7).$$

Note. The node v_0^0 is added to V at the beginning of *collect*. The upper indices $a, \ldots, f$ are not known to the processors and are not used in any protocol. The induced graph appears in Fig. 3.

Let us now explain the edges of G: denote the nodes in REG_1 and REG_2 by v_1^a and v_2^b, respectively. Since $v_1^a.tail\_id = v_0^0.id = 0$ and since $v_1^a.tail\_address = v_0^0.address = 0$, G has an edge from v_0^0 to v_1^a and this edge is denoted by (v_0^0, v_1^a). For the same reason, (v_0^0, v_2^b) is also an edge of G. On the other hand, no node v_i^u satisfies $v_i^u.tail\_id = 1$ and $v_i^u.tail\_address = 1$, thus the outdegree of v_1^a is 0. Using the same rule, the reader can now infer the rest of the edges of G.

Now, we describe the way precedence relations are represented in G: An edge of the precedence tree, (v_j^b, v_i^a), reflects the fact that L_j^b is linearized *before* L_i^a. If two edges (v_i^a, v_j^b) and (v_i^a, v_k^c) emanate

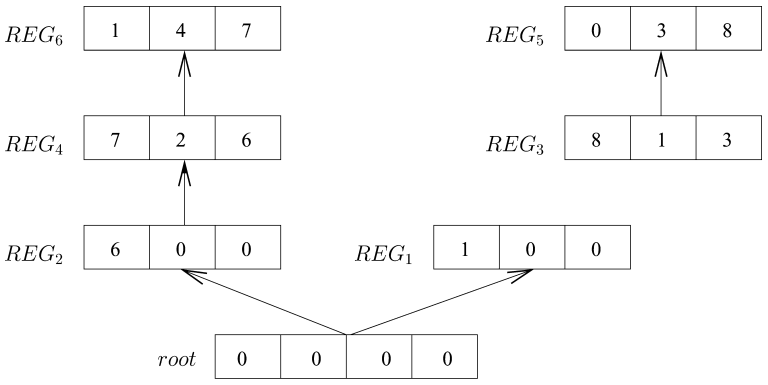

Fig. 3. The induced precedence graph.

from the same node v_i^a, then the order of actions L_j^b and L_k^c cannot be determined by the edges of the precedence tree. In this case, we order these actions by a descending order of their *id*s (higher *id*s are linearized *before* lower *id*s):

Definition 2. Let v_i^a and v_j^b, $i \neq j$, be two nodes with a common tail node. If $i > j$ we say that node v_i^a *locally precedes* node v_j^b.

Note. Relation *locally precedes* is *partial* and it is defined *only* for nodes whose edges emanate from the same node in the precedence graph.

This definition enables linearizing nodes with lower *id*s after nodes with higher *id*s. Using this definition we proceed to define for each precedence graph its *Frontal Branch* and its *last node*:

Definition 3. Let G be some precedence graph. The *frontal branch* of G, denoted by B, is a directed path which is defined inductively as follows:

- The first node in B is the root node v_0^0.
- Let α be a prefix of B and let v_i^a be the last node of α. The next node in B is the node whose edge emanates from v_i^a and which is locally preceded by all other nodes whose edges emanate from v_i^a.
- The last node in the frontal branch of G is called the *last* node of G.

Let $e_1 = (v_k^c, v_i^a)$ and $e_2 = (v_k^c, v_j^b)$ be two edges in some precedence graph G such that e_1 belongs to the frontal branch of G (that is v_j^b locally precedes v_i^a). In this case, we say that e_1 *excludes* e_2 from the frontal branch of G. A pictorial description of a precedence tree appears in Fig. 4, where the edges of the frontal branch appear as bold arrows.

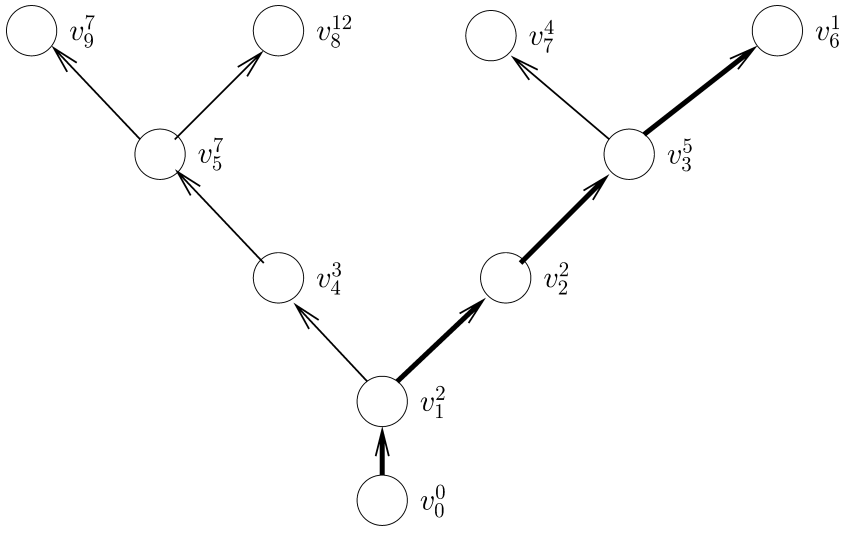

Fig. 4. A precedence tree.

3.3. The sequential implementation

In this section we present a sequential implementation of a (w,r) register. The role of this implementation is to demonstrate the basic ideas, used by all the implementations, in their purest form. In this implementation, each writer writes in a $(1,n)$-register called REG_i, which can be read by all processors, writers and readers; readers do not write. Each register REG_i has a single field that holds the *current* node of $\mathcal{W}_i$. Let c be some system configuration. The *current graph* of c is the graph induced by the root node and the set of all current nodes in c. The basic idea in all the implementations is:

> At any configuration, the last node of the current graph belongs to the write action that was most recently linearized.

Fig. 5 presents the code of the protocols for $\mathcal{W}_i$ (above) and for $\mathcal{R}_u$ (below): The sequential writer protocol works as follows: Logical action L_i^a starts with an invocation of procedure *collect* (see Fig. 2). The graph, returned by *collect*, and its frontal branch are denoted by G_i^a and B_i^a, respectively. The *i-prefix* of B_i^a, denoted by B_i^a/i, is the prefix of B_i^a containing all nodes whose *id* is less then i. Now, $\mathcal{W}_i$ computes a new node, called *new*, that will become its current node at the end of L_i^a. When *new* will be added to the current graph, it will exclude the rest of B_i^a and will become the last node of the new frontal branch. To do that, $\mathcal{W}_i$ chooses the last node of B_i^a/i, denoted by *tail*, as the tail node of *new*, that is: $new.tail\_id := tail.id$ and $new.tail\_address := tail.address$. The address for v_i^a is the minimal address which does not belong to *AD*, thus, the new address is different from component *tail_address* of every current node. In this way, it is ensured that the outdegree of the new node is 0. The sequential reader protocol works as follows: Reader $\mathcal{R}_u$ collects the current graph, computes its frontal branch and returns the last node of the frontal branch.

To complete the definition of the protocol, we need to specify initial values for all data structures. For each writer $\mathcal{W}_i$, REG_i holds its *initial node*, v_i^0, where $v_i^0.address = 0$, $v_i^0.tail\_id = i$ and $v_i^0.tail\_address = 0$. Thus, at the system's initial state all edges of the current graph are self-loops and the initial frontal branch contains solely the root node v_0^0. (This is the only case in the sequential

```
Writer protocol:
begin
        collect(G, AD)
        tail := last node in B_i/i
        new.address := min(AD − AD)
        new.tail_id := tail.id
        new.tail_address := tail.address
        REG_i := write (new)
end

Reader protocol:
begin
        collect(G, AD)
        last := last node in B_u
        return(last)
end
```

Fig. 5. The protocols for $\mathcal{W}_i$ (top) and $\mathcal{R}_u$ (bottom) in the sequential implementation.

implementation in which self-loops are allowed.) The initial logical value is the value corresponding to v_0^0 which can be chosen freely from the permitted values of the logical register.

In Figs. 6 and 7 we demonstrate a single execution of the writer's protocol, assuming that it is invoked in action L_4^3. (*Note*. The index 3 is neither known nor used.) The current graph before L_4^3 starts, appears in Fig. 6, where the edges of the frontal branch are drawn as bold arrows. At the beginning of L_4^3, the current graph G_4^3 is collected. Then, the frontal branch of G_4^3, B_4^3, is computed, and the last node of its 4-*prefix*, $B_4^3/4$, namely, the current node of $\mathcal{W}_1$, is assigned to the variable *tail*. Now, node *new* is computed so that its edge emanates from *tail*. To do this, variables *new.tail_id* and *new.tail_address* are assigned with the *id* and *address* of *tail*, namely 1 and 2, respectively. Following that, the *address* of *new* is chosen as the minimal address not used as *tail_address* in any other current node, namely 1. At this point, computation of *new* is completed and it is written into REG_4 replacing v_4^2 as the *current* node of $\mathcal{W}_4$. The result *current* graph appears in Fig. 7. Note that v_4^3 is the last node of the current graph and v_6^5 is *excluded* from the frontal branch.

Correctness of the sequential implementation is straightforward and is left to the reader. To compute the space complexity, note that component *tail_id* has w possible values, thus, its size is

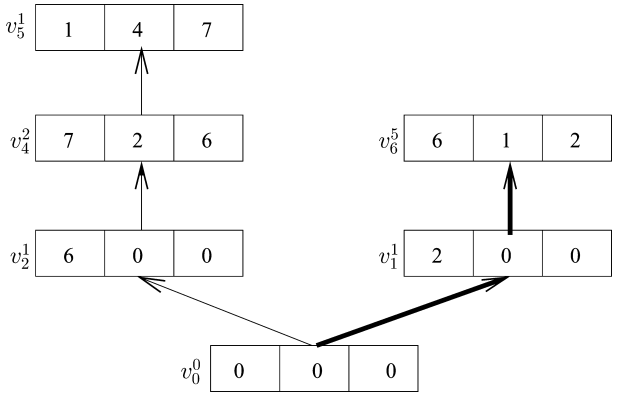

Fig. 6. The sequential writer protocol: before L_4^3.

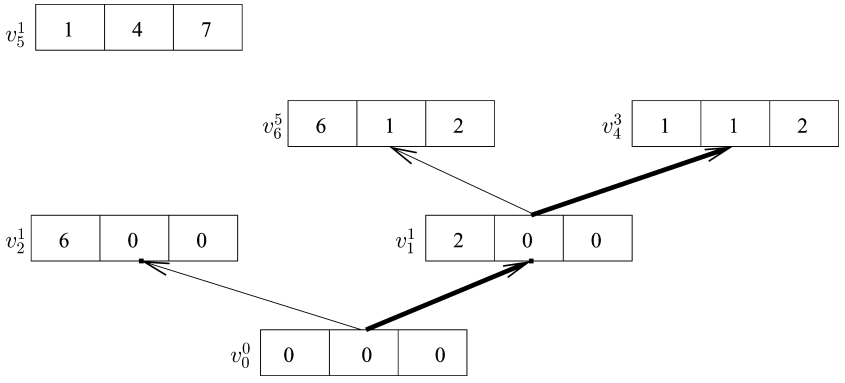

Fig. 7. The sequential writer protocol: after L_4^3.

at most $\log w$. Since the size of the set AD is at most w there should be $w+1$ possible addresses, including 0 for the *root* node, and the size of components *address* and *tail_address* is $\log w + O(1)$. Hence, the protocol's space complexity is $3\log w + O(1)$. The time complexity is straightforward: Every execution of the writer protocol consists of $w+1$ physical actions and every execution of the reader protocol consists of w actions, hence, the time complexity of the sequential implementation is $w + O(1)$. The sequential implementation does not improve upon the complexity of previously known sequential implementations and its role is to present the ideas on which the other two implementations are based.

4. Multi-writer registers using multi-reader registers

In this section, we present a concurrent implementation of a (w,r)-atomic register using physical $(1,n)$-atomic registers. The implementation is obtained by adjusting the sequential implementation to the concurrent environment while preserving the basic ideas and the data structure. In this implementation, communication is again one sided, readers do not write. An important design decision in this implementation is to linearize all logical write actions *independently* of the scheduling of the logical read actions. Under this decision, it holds that at any given configuration, the implemented register has a well-defined value which depends only on the execution of the writers. Linearization of the logical read actions should ensure that every read action returns the register's value at the action's linearization instance. Accordingly, this section is divided into two subsections: The first subsection begins with a description of the writer protocol and the linearization of the logical write actions, and continues with a correctness proof for the linearization. The second subsection begins with a description of the reader protocol and the linearization of the logical read actions, and then proceeds to prove the correctness of the linearization of the read actions. Together the proofs imply the correctness of the entire implementation.

4.1. The writer protocol

4.1.1. Description

In this implementation, once more, each writer has a current node. In every configuration, c, the writers' current nodes induce the *current graph* whose last node is the node of the write action that is linearized last, before c. The value of the last node is the value of the implemented register at c. Concurrency, however, requires a principal difference between the implementations: Correctness of the sequential implementation crucially relies on the fact that procedure *collect* always returns the current graph. In a concurrent environment, the current node of each writer may change many times during a single invocation of *collect* and the collected graph might be different from the current graph of any configuration reached during the invocation. The writer protocol is changed as follows: Node *new* is chosen as in the sequential implementation but it is regarded as tentative and the choice is validated using a declaration mechanism: Writer $\mathcal{W}_i$ declares the tentative *new* node to the other writers by writing it into the *new* field with which REG_i is augmented. Then $\mathcal{W}_i$ rereads the tail node of the edge of *new* and in case the tail node is not changed, node *new* becomes *current*. The writer protocol appears in Fig. 8.

```
begin
    collect (G_i, AD)                                   r_i^a[1], ..., r_i^a[w]
    AD := AD ∪ current.address
    can_tail := last node of B_i/i
    cid := can_tail.id
    new.address := min(AD − AD)
    new.tail_id := cid
    new.tail_address := can_tail.address
    REG_i := write (new, current)                       p_i^a
    tail := read REG_cid.current                        r̃_i^a[cid]
    if can_tail ≃ tail
    then
        /* connect: Direct the edge of v_i^a away from tail */
        current := new
    else
        /* loop: Direct the edge of v_i^a as a self loop */
        current.address := new.address
        current.tail_id := i
        current.tail_address := new.address
    endif
    REG_i := write (new, current)                       w_i^a
end
```

Fig. 8. The protocol for $\mathcal{W}_i$ in the $(1,n)$ implementation.

Now, we describe the protocol in more detail assuming that L_i^a is executed: The register of $\mathcal{W}_i$, REG_i, holds two nodes, denoted by $REG_i.current$ and $REG_i.new$. First $\mathcal{W}_i$ *collects* the graph induced by all *current* nodes. Execution of *collect* takes w atomic physical actions which are denoted by $r_i^a[1]\cdots r_i^a[w]$. Procedure *collect* in this implementation is identical to *collect* in the sequential implementation except that the set *AD* includes all addresses from component *tail_address* of all collected *current* and *new* nodes. Following *collect*, $\mathcal{W}_i$ makes sure that its *new* address will not be equal to its *current* address, by adding $current.address$ $(= v_i^{a-1}.address)$ to the set *AD*. Thus the final size of the set *AD* may reach $2w+2$, including the addresses of the root node (0) and $current.address$.

The graph collected during L_i^a is denoted by G_i^a. The frontal branch of G_i^a is denoted by B_i^a. Let *can_tail* be the last node of B_i^a/i and let *cid* be the *id* of (the owner of) *can_tail*. After G_i^a is collected, $\mathcal{W}_i$ chooses a tentative *new* node whose edge emanates from *can_tail* and whose address is not equal to any address in the set *AD*, thus the outdegree of the new node is 0. In addition, it is ensured that $new.address \neq v_i^{a-1}.address$, thus a writer never uses the same address twice in a row. The chosen node is declared by $\mathcal{W}_i$ by writing it into $REG_i.new$ while $REG_i.current$ is not changed. The declaring write action is denoted by p_i^a.

Following p_i^a, $\mathcal{W}_i$ rereads $REG_{cid}.current$. This second read action is denoted by $\tilde{r}_i^a[cid]$. After $\tilde{r}_i^a[cid]$, $\mathcal{W}_i$ computes v_i^a. The following notation is needed in order to describe the way v_i^a is computed: Let v_j^b and v_j^c be two nodes that are equal as records, namely: $v_j^b.address = v_j^c.address$, $v_j^b.tail\_id = v_j^c.tail\_id$ and $v_j^b.tail\_address = v_j^c.tail\_address$. The fact that v_j^b and v_j^c are equal (as records) is denoted by $v_j^b \simeq v_j^c$.

Note. The fact that $v_j^b \simeq v_j^c$ does not imply that $b = c$.

The (*current*) node of L_i^a, v_i^a, is chosen as follows: Let *can_tail* and *tail* be the two nodes read by $\mathcal{W}_i$ in actions $r_i^a[cid]$ and $\tilde{r}_i^a[cid]$, respectively. If *can_tail* $\simeq$ *tail* then the *new* node is assigned to *current*, in this case we say that L_i^a *connects*. If, however, *can_tail* $\not\simeq$ *tail* then *current* is chosen so that its edge is a self-loop directed from v_i^a towards itself. In this case, we say that L_i^a *loops*. The logical write action L_i^a is concluded by a final physical write action in which $\mathcal{W}_i$ writes v_i^a in REG_i replacing v_i^{a-1} as its *current* node. This last concluding write action is denoted by w_i^a.

To complete the definition of the protocol, we need to specify initial values for all data structures. Similar to the sequential implementation, both the *current* and the *new* nodes are initialized to hold the initial node, v_i^0, where $v_i^0.address = 0$, $v_i^0.tail\_id = i$ and $v_i^0.tail\_address = 0$. That is, at the initial configuration all edges of the current graph are self-loops, its initial frontal branch contains only the root node v_0^0 and the initial logical value is the value corresponding to v_0^0, which can be chosen freely from the set of all permitted values.

Component *tail_id* has $w + 1$ possible values and its size is $\log w + O(1)$. Since the size of the set *AD* is at most $2w + 2$ there should be $2w + 3$ possible addresses and the size of components *address* and *tail_address* is also $\log w + O(1)$. Thus, the space required by each node is $3 \log w + O(1)$ and since the each register contains 2 nodes, the protocol's space complexity is $6 \log w + O(1)$. The time complexity of the writer's protocol in this implementation is $w + 3$, assuming that each writer reads its own register.

4.1.2. Linearization of the logical write actions

Now, we fix some arbitrary execution of the system, $E = c_0, d_1, c_1, \ldots$, and all definitions and proofs are made with respect to E. Since E is arbitrary, the results hold for every system execution of the implementation. The logical write actions are linearized using a *History graph*—a precedence graph which reflects the execution of the system. The history graph, which is not computable by the processors, plays a key role in the correctness proof of our protocols.

Definition 4. Let v_i^a be the node of logical write action L_i^a. The *tail node* of v_i^a is defined as follows:

(1) If L_i^a connects then the tail node of v_i^a is the node *tail*, read in $\tilde{r}_i^a[cid]$, which is $\simeq$ to *can_tail* (read in $r_i^a[cid]$).
(2) If L_i^a loops then the tail node of v_i^a is v_i^a itself.

Definition 5. Let $E = c_0, d_1, c_1, \ldots$ be an execution of the system. For every configuration c_t of E, the *History graph* of c_t, H^{c_t} (denoted also as H^{d_t}), is defined as follows:

(1) H^{c_0} is the *History graph* before the execution begins; it contains only the *root* node—v_0^0.
(2) Let c_t be an arbitrary configuration of E. If $d_t = w_i^a$, for some i and a, then $H^{c_t} = H^{w_i^a}$ is obtained from $H^{c_{t-1}}$ by adding node, v_i^a, and an edge directed from the *tail node* of v_i^a into v_i^a. If $d_t \neq w_i^a$ for any i and a then $H^{c_t} = H^{c_{t-1}}$.

For any configuration c_t, the history graph H^{c_t} is a precedence forest which consists of a single precedence tree and some disjoint self-loops. The number of nodes in H^{c_t} is equal to the number of logical write actions completed until c_t plus one (for the root node). It should be noted that though each node of H corresponds to some node of the precedence graph $\mathcal{P}$, the graph H is not a

subgraph of $\mathcal{P}$. Before we define the frontal branch of H, we should extend relation *locally precedes* to accommodate the following situation: Unlike the collected precedence graphs, the graph H may have a node v_k^c with several edges directed from v_k^c into several nodes of the same writer. Since the actions of each individual processor are temporally ordered by their indices, nodes with equal *id*s are ordered by their indices where a node with a lower index *locally precedes* a node with a higher index.

Definition 6. Let v_i^a and v_j^b be two nodes with a common tail node. Node v_i^a *locally precedes* node v_j^b if either $i > j$ or if $i = j$ and $a < b$.

Under the new definition of *locally precedes*, the frontal branch of H^{d_t}, denoted by $B_H^{d_t}$, is defined just as before in Definition 3. The next definition is needed for the linearization of write actions:

Definition 7. Let L_i^a be an arbitrary logical write action. Action L_i^a is *lasting* if v_i^a is last in $B_H^{w_i^a}$. If L_i^a is not lasting, it is *transient*.

Obviously if L_i^a loops then it is transient, but there are many cases in which L_i^a connects but it is still transient. For example, consider a situation in which the last node of the current graph is v_i^a and write actions L_j^b and L_k^c, where $i < j < k$, are executed, while all other writers and readers are idle. The physical actions of L_j^b and L_k^c are executed as follows: first, all physical actions of L_j^b except w_j^b are executed. Then all physical actions of L_k^c except w_k^c are executed. In this situation, both writers collect the same precedence graph (which is equal to the *current* graph), choose v_i^a as tail, compute a *new* node whose edge emanate from v_i^a, declare the new node to $\mathcal{W}_i$ (actions p_j^b and p_k^c), reread REG_i (actions $\tilde{r}_j^b[i]$ and $\tilde{r}_k^c[i]$) and connect. Now, assume that w_j^b is executed. At this instance, the edge (v_i^a, v_j^b) is added to H, and the last node of H is v_j^b, so L_j^b is lasting. Following that w_k^c is executed and the edge (v_i^a, v_k^c) is added to H. At this instance, however, the edge (v_i^a, v_j^b) is already in H so v_j^b excludes v_k^c from $B_H^{w_k^c}$, that is: L_k^c is transient.

Using the partition of logical write actions to lasting and transient we linearize all the logical write actions. For every execution $E = c_0, d_1, c_1, \ldots$, linearization points can be specified by E's configurations or by its physical actions. When we say that logical action a is linearized at physical action d_i we mean that the linearization point is after d_i has taken its effect and actually it can be looked at as if a is linearized at configuration c_i.

Definition 8. Let L_i^a be an arbitrary logical write action: The *linearization point* of L_i^a is defined as follows:

(1) If L_i^a is lasting then L_i^a is linearized at w_i^a.
(2) If L_i^a is transient and v_j^b is the last node of $B_H^{w_i^a}$ then L_i^a is linearized before w_j^b and after any other physical action which precedes w_j^b. In this case, we say that L_i^a is *linearized by* L_j^b.
(3) If two transient write actions, L_i^a and L_j^b, are linearized by the same lasting write action L_k^c, L_i^a and L_j^b are linearized by ascending order of their *id*s.

Under the defined linearization, the value written by a lasting logical write, L_i^a, is the value of the implemented register from the point of execution of w_i^a until the next lasting logical write

completes its execution, or an infinitesimally short time *before* the next lasting logical write completes its execution. On the other hand, the value written by a transient logical write is the value of the implemented register only for an infinitesimally short period.

4.1.3. Correctness of the linearization of logical write actions

In the correctness proof, we have to prove that the linearization point satisfies the linearization requirements for atomic registers. In Theorem 10 we prove that every logical write action is linearized within its execution interval. In Theorem 11 we prove that all graphs collected by the writers are precedence graphs. We start the correctness proof with some technical lemmas.

Lemma 1. *Let d_t be an arbitrary physical action and let v_i^a be an arbitrary node in H^{d_t}. If $v_i^a \notin B_H^{d_t}$, then for every $u > t$, $v_i^a \notin B_H^{d_u}$.*

Proof. If L_i^a loops then for any $u \geqslant t$, v_i^a is not connected to the root in H^{d_u}. Assume that L_i^a connects and denote the path from the root to v_i^a, in H^{d_t}, by α. Since $v_i^a \notin B_H^{d_t}$, let (v_j^b, v_k^c) be the first edge which is in α but not in $B_H^{d_t}$ and let (v_j^b, v_ℓ^d) be the edge excluding (v_j^b, v_k^c) from $B_H^{d_t}$. No edge of H is ever deleted, therefore, for any $u > t$, (v_j^b, v_ℓ^d) excludes the suffix of α and in particular v_i^a, from $B_H^{d_u}$. □

Let d_t and d_u be two physical atomic actions; the fact that d_t occurs before d_u, i.e. $t < u$, is denoted by $d_t \to d_u$.

Lemma 2. *If (v_j^b, v_i^a), $i \neq j$, is an edge of H then w_j^b occurs before w_i^a (i.e., $w_j^b \to w_i^a$).*

Proof. According to the definition of the history graph, (v_j^b, v_i^a), $i \neq j$, is an edge of H only if the *id* of the last node of B_i^a/i is equal to j, and the tail node of v_i^a (that is the node read in $\tilde{r}_i^a[cid]$) is v_j^b. Therefore $w_j^b \to \tilde{r}_i^a[j]$. Since $\tilde{r}_i^a[j] \to w_i^a$, we get $w_j^b \to w_i^a$. □

Lemma 3. *If, for some $t > 0$, $v_i^a \in B_H^{d_t}$ then L_i^a is lasting.*

Proof. By Lemma 1, $v_i^a \in B_H^{w_i^a}$. Lemma 2 implies that for any edge (v_i^a, v_j^b) in H, $w_i^a \to w_j^b$. Hence the outdegree of v_i^a in $H^{w_i^a}$ is 0, and v_i^a is last in $B_H^{w_i^a}$. By Definition 7, L_i^a is lasting. □

Lemma 4.

(*a*) *The sequence of node ids along any directed path of the history graph, from the root towards any leaf, is strictly increasing.*
(*b*) *Every directed path of the history graph contains at most one node of every writer.*

Proof. According to the protocol, the tail node of v_i^a is $\simeq$ to the last node of B_i^a/i. Since B_i^a/i contains only nodes whose $id < i$, the proof for (a) follows. (b) is implied immediately by (a). □

The next four (quite technical) lemmas deal with the relations between the history graph H, its frontal branch B_H, trees collected by the writers and the writers' current nodes. These lemmas are used in the proof of Theorem 10 in which correctness of the linearization is proved. The next lemma demonstrates a principal difference between the sequential implementation and the concurrent implementation: In the sequential implementation, whenever a processor reads nodes of two

processors, say v_j^b and v_i^a ($j < i$), and discovers that there is an edge from v_j^b to v_i^a, it can immediately conclude that the edge (v_i^a, v_j^b) is part of the current graph. In concurrent environment, the situation is very different. For example, $\mathcal{W}_k$ may read $v = v_j^b$, then $\mathcal{W}_j$ may compute several nodes until, in L_j^{b+r}, it reaches v again, that is $v_j^b \simeq v_j^{b+r}$. At this point, $\mathcal{W}_i$ may execute L_i^a and compute v_i^a so that its edge is (v_j^{b+r}, v_i^a). Now, if $\mathcal{W}_k$ reads v_i^a, the edge (v_j^b, v_i^a) is in G_k^c while the current graph (and the history graph H) has the edge (v_j^{b+r}, v_i^a). In the following lemma we prove that whenever (v_j^b, v_i^a) is in G_k^c, either it is also in H or the cause for the edge of G_k^c is the scenario described above.

Lemma 5. *If (v_j^b, v_i^a) is an edge of G_k^c then there exists an integer r, $r \geqslant 0$, such that (v_j^{b+r}, v_i^a) is an edge of $H^{w_i^a}$.*

Proof. By the writer's protocol code, $j \leqslant i$. If $j = i$ then L_i^a loops and the lemma follows immediately. We continue the proof assuming that $j < i$, that is L_i^a connects. First, we prove that if (v_j^b, v_i^a) is an edge of G_k^c then $r_j^b[i] \to p_i^a$. Assume by way of contradiction that $p_i^a \to r_j^b[i]$; by the protocol $r_j^b[i] \to w_j^b$. Since v_j^b is a node in G_k^c, it holds that $w_j^b \to r_k^c[j]$. Since G_k^c is collected in ascending order, it holds that $r_k^c[j] \to r_k^c[i]$. Thus we get that $p_i^a \to r_j^b[i] \to r_k^c[i]$. Since $v_i^a \in G_k^c$, v_i^a is the *current* node of $\mathcal{W}_i$ at $r_k^c[i]$, so we can conclude that during the interval $[p_i^a\ ,\ r_k^c[i]]$, v_i^a is either the *new* node of $\mathcal{W}_i$ or it is its *current* node. Therefore, $v_i^a.tail\_address \in AD_j^b$, where AD_j^b is the set AD returned by procedure *collect* invoked during L_j^b. In this case, the code implies that $v_j^b.address \neq v_i^a.tail\_address$. On the other hand, the fact that (v_j^b, v_i^a) is an edge of G_k^c implies that $v_i^a.tail\_address = v_j^b.address$, a contradiction.

Now, we continue the proof assuming $r_j^b[i] \to p_i^a$. Since L_i^a connects, Definition 5 implies that there exists some integer r such that v_j^{b+r} is the current node of $\mathcal{W}_j$ at $\tilde{r}_i^a[j]$ and (v_j^{b+r}, v_i^a) is an edge of H. To prove the lemma we have to show that $r \geqslant 0$. Assume by way of contradiction that $r < 0$. Since (v_j^b, v_i^a) is an edge of G_k^c, $v_i^a.tail\_address = v_j^b.address$; since (v_j^{b+r}, v_i^a) is an edge of H, $v_i^a.tail\_address = v_j^{b+r}.address$; thus $v_j^b.address = v_j^{b+r}.address$. Since a writer never uses the same address twice in a row, we conclude that $r < -1$ and $w_j^{b+r} \to w_j^{b-1} \to r_j^b[i]$. Since $r_j^b[i] \to p_i^a$, we get that $w_j^{b+r} \to w_j^{b-1} \to r_j^b[i] \to p_i^a \to \tilde{r}_i^a[j]$, and in particular $w_j^{b+r} \to w_j^{b-1} \to \tilde{r}_i^a[j]$. Therefore, v_j^{b+r} is not the current node of $\mathcal{W}_j$ at $\tilde{r}_i^a[j]$, and the edge (v_j^{b+r}, v_i^a) does not belong to H, a contradiction. □

In the next lemma, we prove that if v_i^a belongs to the frontal branch of the history graph in some arbitrary configuration c_t then v_i^a is the current node of $\mathcal{W}_i$ in c_t. In the proof we use the following notation: The fact that v_i^a and v_j^b are actually the same node, i.e., $i = j$ and $a = b$, is denoted by $v_i^a \equiv v_j^b$. The fact that every node $v_j^b \in B_H^{c_t}$ is in B_i^a and every node $v_k^c \in B_i^a$ is in $B_H^{c_t}$ is denoted by $B_H^{c_t} \equiv B_i^a$.

Lemma 6. *If $v_i^a \in B_H^{c_t}$ for some configuration c_t, then v_i^a is the current node of $\mathcal{W}_i$ at c_t.*

Proof. Assume by way of contradiction that there are nodes which do not satisfy the lemma. Let v_i^a be the first node in E among these nodes, that is, there exists some configuration c_t, such that w_i^{a+1} occurs before c_t, while $v_i^a \in B_H^{c_t}$. Under these conditions, Lemma 1 implies that $v_i^a \in B_H^{w_i^{a+1}}$. By the same Lemma $v_i^a \in B_H^{w_i^a}$, that is $B_H^{w_i^a}/i \equiv B_H^{w_i^{a+1}}/i$. By the minimality of v_i^a we get that all nodes in $B_H^{w_i^{a+1}}/i$ are the current nodes of their owners throughout L_i^{a+1}, Thus $B_H^{w_i^{a+1}}/i$ is a subgraph of G_i^{a+1}.

Now we prove that $B_i^{a+1}/i \equiv B_H^{w_i^{a+1}}/i$. Assume towards a contradiction that $B_i^{a+1}/i \not\equiv B_H^{w_i^{a+1}}/i$. Since $B_H^{w_i^{a+1}}$ is a subgraph of G_i^{a+1}, there exists a node of $B_H^{w_i^{a+1}}/i$ (and of B_i^{a+1}/i), v_j^b, and an edge of G_i^{a+1}, (v_j^b, v_k^c) such that $j < k < i$, and the edge (v_j^b, v_k^c) excludes the rest of $B_H^{w_i^{a+1}}$ from B_i^{a+1}/i. Since $v_j^b \in B_H^{w_i^{a+1}}/i$, and (v_j^b, v_k^c) is an edge of G_i^{a+1}, Lemma 5 implies that in $H^{w_k^c}$ there exists an edge (v_j^{b+r}, v_k^c), for some $r \geqslant 0$. By the minimality of v_i^a we get that v_j^b is the current node of $\mathcal{W}_j$ throughout L_i^{a+1}. Since v_k^c is a node of G_i^a we conclude that v_j^b is the current node of $\mathcal{W}_j$ throughout L_k^c and in particular, the node read in $\tilde{r}_k^c[j]$. By Definition 5, we conclude that (v_j^b, v_k^c) is an edge of $H^{w_i^{a+1}}$. This, however, means that the edge (v_j^b, v_k^c) excludes the rest of $B_H^{w_i^a}/i$ from $B_H^{w_i^{a+1}}/i$, hence $v_i^a \notin B_H^{w_i^{a+1}}$, a contradiction. We conclude $B_i^{a+1}/i \equiv B_H^{w_i^{a+1}}/i$.

Since $B_i^{a+1}/i \equiv B_H^{w_i^{a+1}}/i$, L_i^{a+1} connects and v_i^{a+1} is last in $B_H^{w_i^{a+1}}$. This, however, means that v_i^{a+1} excludes v_i^a from $B_H^{w_i^{a+1}}$, in contradiction to the assumption that $v_i^a \in B_H^{c_t}$. The lemma follows. □

Lemma 7. *If L_i^a is transient then the last node of $B_H^{w_i^a}/i$ is $\not\equiv$ to the last node of B_i^a/i and hence, $B_H^{w_i^a}/i \not\equiv B_i^a/i$.*

Proof. Let v_j^b be the last node of $B_H^{w_i^a}/i$ and assume by way of contradiction that v_j^b is also the last node of B_i^a/i. Since v_j^b is a node in G_i^a, we conclude that $w_j^b \to r_i^a[j]$. Since v_j^b is a node in $B_H^{w_i^a}$, Lemma 6 implies that v_j^b is the current node of $\mathcal{W}_j$ at w_i^a, hence v_j^b is the current node of $\mathcal{W}_j$ during the interval $[r_i^a[j], w_i^a]$ and in particular at $\tilde{r}_i^a[j]$. Therefore, nodes *can_tail* and *tail* read during L_i^a are both $\equiv$ to v_j^b, hence L_i^a connects. Since v_j^b is last in $B_H^{w_i^a}/i$, v_i^a is last in $B_H^{w_i^a}$, that is L_i^a is lasting, a contradiction. The lemma follows. □

Though in general edges of collected graphs are not in H, the next corollary gives a sufficient condition for a collected edge to be in H:

Corollary 8. *Let (v_j^b, v_i^a) be an edge of G_k^c. If for some configuration c_t after $r_k^c[i]$, $v_j^b \in B_H^{c_t}$ then (v_j^b, v_i^a) is an edge of H^{c_t}.*

Proof. Since (v_j^b, v_i^a) is an edge of G_k^c, Lemma 5 implies that for some r, $r \geqslant 0$, (v_j^{b+r}, v_i^a) is an edge of H^{c_t}. Since $v_j^b \in B_H^{c_t}$, Lemma 6 implies that v_j^b is the current node of $\mathcal{W}_j$ at c_t, therefore $r = 0$. □

In the next lemma and in the theorem that follows, we prove that the linearization point of each logical write action lies within its execution interval.

Lemma 9. *If for some configuration c_t, after $r_i^a[j]$, it holds that $B_H^{c_t}/(j+1) \not\equiv B_i^a/(j+1)$, then there exists a lasting logical write action L_k^c, for some $k \leqslant j$, such that w_k^c occurs within the interval starting at $r_i^a[k]$ and ending at c_t.*

Proof. By the assumption of the lemma, $B_H^{c_t}/(j+1) \not\equiv B_i^a/(j+1)$. Let v_ℓ^d, $\ell \leqslant j$, be the node with the maximal *id* in the common prefix of $B_H^{c_t}/(j+1)$ and $B_i^a/(j+1)$. Such a node always exists because the root, v_0^0, belongs to both branches.

First, we show that v_ℓ^d is not the last node of $B_H^{c_t}/(j+1)$ and identify action L_k^c: if v_ℓ^d is the last node of $B_H^{c_t}/(j+1)$ then since $B_H^{c_t}/(j+1) \not\equiv B_i^a/(j+1)$, v_ℓ^d is not the last node of $B_i^a/(j+1)$. Let $e_1 = (v_\ell^d, v_p^f)$, $p < j+1$, be the edge emanating from v_ℓ^d in $B_i^a/(j+1)$. Since $v_\ell^d \in B_H^{c_t}$, Corollary 8 implies that (v_ℓ^d, v_p^f) is an edge of H^{c_t}; since $p < j+1$, either $e_1 \in B_H^{c_t}/(j+1)$ or there exists some edge $e_2 \in B_H^{c_t}/(j+1)$ excluding e_1 from $B_H^{c_t}/(j+1)$, a contradiction to the assumption that v_ℓ^d is the last node of $B_H^{c_t}/(j+1)$. Since v_ℓ^d is not the last node of $B_H^{c_t}/(j+1)$, let (v_ℓ^d, v_k^c), $\ell < k \leqslant j$, be the edge emanating from v_ℓ^d in $B_H^{c_t}/(j+1)$.

Now, we show that L_k^c satisfies the requirements of the lemma: The situation is depicted in Fig. 9. Since $v_k^c \in B_H^{c_t}$, Lemma 3 implies that L_k^c is lasting. Since $v_k^c \in H^{c_t}$, w_k^c occurs before c_t. To complete the proof it remains to show that $r_i^a[k] \to w_k^c$. Assume by way of contradiction that $w_k^c \to r_i^a[k]$. Now we show that (v_ℓ^d, v_k^c) is an edge of G_i^a (but not of $B_i^a/(j+1)$). By its definition, v_k^c is a node of $B_H^{c_t}$, therefore, by Lemma 6, v_k^c is the current node of $\mathcal{W}_k$ at c_t. By our assumption $w_k^c \to r_i^a[k]$, hence v_k^c is a node of G_i^a and (v_ℓ^d, v_k^c) is an edge of G_i^a. Since $k > \ell$, and v_ℓ^d is the maximal common node in $B_H^{c_t}/(j+1)$ and $B_i^a/(j+1)$ we conclude that $v_k^c \notin B_i^a/(j+1)$. Let (v_ℓ^d, v_m^e), $\ell < m < k$, be the edge excluding (v_ℓ^d, v_k^c) from $B_i^a/(j+1)$. By Corollary 8, (v_ℓ^d, v_m^e) is an edge in H^{c_t}. Since $m < k$, (v_ℓ^d, v_m^e) excludes (v_ℓ^d, v_k^c) from $B_H^{c_t}$, a contradiction to the assumption that (v_ℓ^d, v_k^c) is in $B_H^{c_t}/(j+1)$. The lemma follows. □

Theorem 10. *Every logical write action is linearized within its execution interval.*

Proof. Let L_i^a be some logical write action. If L_i^a is lasting then it is linearized at its concluding physical write w_i^a, which is within its execution interval. For the rest of the proof we assume that L_i^a is transient. Let v_j^b be the last node of $B_H^{w_i^a}$ by which L_i^a is linearized. We have to prove that L_j^b is linearized within the execution interval of L_i^a, which follows if we show $r_i^a[1] \to w_j^b \to w_i^a$. Since $v_j^b \in H^{w_i^a}$, it is clear that $w_j^b \to w_i^a$. Now, we prove that $r_i^a[1] \to w_j^b$. Action L_i^a is transient, therefore, Lemma 7 implies that $B_H^{w_i^a}/i \not\equiv B_i^a/i$. By Lemma 9 (with $j = i-1$), there exists a lasting logical write action L_k^c, $1 \leqslant k < i$, such that w_k^c occurs within the interval starting at $r_i^a[k]$ and ending at w_i^a. For

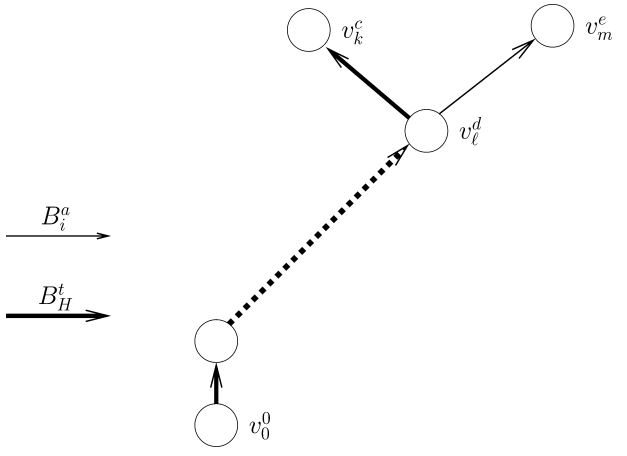

Fig. 9. The graph for Lemma 9.

$k = 1$, we get $r_i^a[1] \to w_k^c \to w_i^a$. For $k > 1$ we use the fact that $r_i^a[1] \to r_i^a[k]$, and once again get $r_i^a[1] \to w_k^c \to w_i^a$. If $L_j^b \equiv L_k^c$ then the proof follows. Otherwise, since L_j^b and L_k^c are both lasting, and since v_j^b is the last node of $B_H^{w_i^a}$, we get that $w_k^c \to w_j^b$. Therefore, $r_i^a[1] \to w_k^c \to w_j^b$, which implies $r_i^a[1] \to w_j^b$. □

The fact that L_i^a is linearized before L_j^b is denoted by $L_i^a \Rightarrow L_j^b$. The next theorem says that the graphs collected by the processors are precedence graphs with respect to the relation $\Rightarrow$.

Theorem 11.

(*a*) *If* (v_j^b, v_i^a), $j < i$, *is an edge of* H *then* $L_j^b \Rightarrow L_i^a$.
(*b*) *If* (v_j^b, v_i^a), $j < i$, *is an edge of* G_k^c *then* $L_j^b \Rightarrow L_i^a$.

Proof of (a). By Lemma 2, $w_j^b \to w_i^a$. Consider the following cases:

Case 1: L_i^a *is lasting.* In this case, L_i^a is linearized at w_i^a. Since L_j^b is not linearized after w_j^b, Lemma 2 implies that $w_j^b \to w_i^a$, hence $L_j^b \Rightarrow L_i^a$.

Case 2: L_i^a *and* L_j^b *are both transient.* Let v_k^c and v_ℓ^d be the last nodes of $B_H^{w_i^a}$ and $B_H^{w_j^b}$, respectively. In this case L_i^a is linearized by L_k^c and L_j^b is linearized by L_ℓ^d. If $v_k^c \equiv v_\ell^d$, then by Definition 8, L_i^a and L_j^b are linearized by their *id*s. Since $i > j$, $L_j^b \Rightarrow L_i^a$. Assume $v_k^c \not\equiv v_\ell^d$. By Definition 8, L_ℓ^d is the last lasting write action linearized before w_j^b and L_k^c is the last lasting write action linearized before w_i^a. Since $w_j^b \to w_i^a$ it holds that $w_\ell^d \to w_k^c$ and therefore $L_\ell^d \Rightarrow L_k^c$. Since L_j^b is linearized by L_ℓ^d and L_i^a is linearized by L_k^c, we get $L_j^b \Rightarrow L_i^a$.

Case 3: L_i^a *is transient,* L_j^b *is lasting.* First we show that L_i^a is not linearized by L_j^b. If this was the case, v_j^b would have been last in $B_H^{w_i^a}$, but this means that L_i^a is lasting, a contradiction. The rest of the proof of this case is very similar to the proof of the previous case.

Proof of (b). By Lemma 5, there exists some $r, r \geqslant 0$, such that (v_j^{b+r}, v_i^a) is an edge of H. By (a), $L_j^{b+r} \Rightarrow L_i^a$. If $r = 0$ then we get $L_j^b \Rightarrow L_i^a$. If $r > 0$ then, since $L_j^b \Rightarrow L_j^{b+r}$, we get again that $L_j^b \Rightarrow L_i^a$. □

4.2. The reader protocol

4.2.1. Description

The basic idea in this implementation is to maintain a precedence graph whose last node at any configuration c_t belongs to the logical write linearized last before c_t. Unfortunately, it is not sufficient for a reader to just collect a precedence graph and return the last node in the graph's frontal branch: Due to concurrency, the current graph and its last node may change during the execution of *collect* by the reader. As a result, a node of a logical write action which should not be returned by a reader may appear as last in the reader's collected graph.

To overcome this problem, the reader collects three graphs; the first and third graphs are collected by procedure *collect* of the writer's protocol and they are denoted by G and $\hat{G}$, respectively. The second graph, denoted by $\overleftarrow{G}$, is collected by procedure $\overleftarrow{collect}$ which is identical to *collect* except

begin
  $collect(G_u, AD)$; $\overleftarrow{collect}$ $(\overleftarrow{G}_u, AD)$; $collect(\hat{G}_u, AD)$
  if $(B_u \simeq \hat{B}_u \not\simeq \overleftarrow{B}_u))$ **then**
    $\ell := \min_j(v_j^b (\in G_u) \not\simeq v_j^c (\in \overleftarrow{G}_u)$ **and** $v_j^c \not\simeq v_j^d (\in \hat{G}_u))$
    return v_ℓ^d
  elseif $(B_u \simeq \overleftarrow{B}_u \simeq \hat{B}_u)$ **then**
    return the *last* node of $\hat{B}_u$
  elseif $(B_u \not\simeq \hat{B}_u)$ **and** $(B_u \subset \hat{G}_u)$ **then**
    return the *last* node of $\hat{B}_u$
  elseif $(B_u \not\simeq \hat{B}_u)$ **and** $(B_u \not\subset \hat{G}_u)$ **then**
    $i := \min_j(v_j^a (\in B_u) \not\simeq v_j^b (\in \hat{G}_u))$
    return v_i^b
  endif
end

Fig. 10. The protocol for $\mathcal{R}_u$ in the $(1, n)$ implementation.

that nodes are collected in reverse order—from REG_W down to REG_1 (therefore most of the lemmas proven in the previous section do not hold for $\overleftarrow{G}$). After G, $\overleftarrow{G}$ and $\hat{G}$ are collected, they are analyzed and a node satisfying the requirements for the reader protocol is identified and returned. The code of the reader protocol appears in Fig. 10.

Let $\mathcal{R}_u$ be a reader executing the protocol. The physical actions, executed by $\mathcal{R}_u$, during the reader protocol are denoted by: $r_u[1] \cdots r_u[w]$, in which G is collected, $\overleftarrow{r}_u[w] \cdots \overleftarrow{r}_u[1]$, in which $\overleftarrow{G}$ is collected, and $\hat{r}_u[1] \cdots \hat{r}_u[w]$, in which $\hat{G}$ is collected. The notation $B_i^a \subset B_j^b$ is used when for each $v_k^c \in B_i^a$, there is a node $v_k^d \in B_j^b$ such that $v_k^c \simeq v_k^d$. The notation $B_i^a \simeq B_j^b$ is used when $B_i^a \subset B_j^b$ and $B_j^b \subset B_i^a$.

4.2.2. Linearization of the logical read actions

Throughout this paper S_u^a denotes the ath execution of the read protocol by $\mathcal{R}_u$. The linearization point of any logical read action is determined by the linearization point of the logical write action whose value is returned by the read action, as follows.

Definition 9. Let v_i^b be the node returned by S_u^a. Denote by c_s and c_t the result configurations of $r_u^a[1]$ and $\hat{r}_u^a[w]$, respectively. The linearization point of S_u^a is defined as follows:

(1) If L_i^b is linearized before c_s then S_u^a is linearized at c_s.
(2) If L_i^b is linearized after c_s then S_u^a is linearized by L_i^b, that is after L_i^b and before any physical action which occurs after L_i^b (or before any logical write which is linearized after L_i^b). In case that two logical read actions are linearized by the same logical write action, they are linearized by an ascending order of their *id*s.

4.2.3. Correctness of the implementation

The correctness of the linearization scheme for logical read actions, and the correctness of the entire implementation, is proved by the following theorem.

Theorem 12. *Let (c_s, c_t) be the execution interval of S_u, let v be the node returned by S_u, and let L be the logical write action that produced the node v. Logical action L satisfies one of the following two claims*:

(1) *Either*
L is the last logical write action linearized before c_s(and hence v is last in $B_H^{c_s}$).
(2) *Or*
L is linearized within the interval (c_s, c_t).

Proof. Since v is read during the execution of S_u, clearly L terminates before c_t. Therefore, to conclude that (2) holds, it suffices to show that L is linearized after c_s. Consider the following cases (which match the cases of the protocol):

Case 1: $B_u \simeq \hat{B}_u \not\simeq \overleftarrow{B}_u$.
Let ℓ be the smallest *id* such that $v_\ell^b \not\simeq v_\ell^c$ and $v_\ell^c \not\simeq v_\ell^d$, where v_ℓ^b, v_ℓ^c and v_ℓ^d are nodes in G_u, $\overleftarrow{G}_u$, and $\hat{G}_u$, respectively. By Claim 1 such nodes always exist. According to the protocol S_u returns v_ℓ^d. Since $v_\ell^b \not\simeq v_\ell^c$ we get $b < c$ and $r_u[1] \to r_u[\ell] \to w_\ell^c$. Since $v_\ell^c \not\simeq v_\ell^d$ we get $c < d$ and $w_\ell^c \to r_\ell^d[1]$. We conclude that $r_u[1] \to r_\ell^d[1]$, hence L_ℓ^d is linearized after c_s.

Claim 1. *Under the conditions of Case* 1, *there exists an integer $\ell, 1 \leqslant \ell \leqslant w$, such that $v_\ell^b \not\equiv v_\ell^c \not\equiv v_\ell^d$, where v_ℓ^b, v_ℓ^c and v_ℓ^d are nodes in G_u, $\overleftarrow{G}_u$, and $\hat{G}_u$, respectively.*

Proof of claim. By the conditions for Case 1 we have $B_u \simeq \hat{B}_u \not\simeq \overleftarrow{B}_u$. Consider the following cases:

Case 1.1: B_u is a subgraph of $\overleftarrow{G}_u$.
Let v_k^a be the last node on the common prefix of B_u and $\overleftarrow{B}_u$ and let (v_k^a, v_ℓ^c) be the first edge of $\overleftarrow{B}_u$ not included in B_u. Such an edge always exists since, by the conditions of Cases 1 and 1.1, we have $B_u \not\simeq \overleftarrow{B}_u$ and $B_u \subset \overleftarrow{G}_u$. Let v_ℓ^b, v_ℓ^c and v_ℓ^d be the nodes of $\mathcal{W}_\ell$ in G_u, $\overleftarrow{G}_u$, and $\hat{G}_u$, respectively. Obviously $v_\ell^b \not\simeq v_\ell^c$ because otherwise the edge (v_k^a, v_ℓ^c) would have been included in B_u. Since $B_u \simeq \hat{B}_u$ we also have $v_\ell^c \not\simeq v_\ell^d$. The claim follows.

Case 1.2: B_u is not a subgraph of $\overleftarrow{G}_u$.
Let v_ℓ^b be the first node on B_u which does not belong to $\overleftarrow{B}_u$. Such a node always exists because B_u is not a subgraph of $\overleftarrow{G}_u$. Let v_ℓ^b, v_ℓ^c and v_ℓ^d be the nodes of $\mathcal{W}_\ell$ in G_u, $\overleftarrow{G}_u$, and $\hat{G}_u$, respectively. Obviously $v_\ell^b \not\simeq v_\ell^c$ and $v_\ell^c \not\simeq v_\ell^d$. The claim follows. □

Case 2: $B_u \simeq \hat{B}_u \simeq \overleftarrow{B}_u$.

Let v_ℓ^d be the last node of $\hat{B}_u$. According to the protocol S_u returns v_ℓ^d. Consider the following cases:

Case 2.1: $B_u \not\equiv \hat{B}_u$.
In this case, there exist two distinct nodes v_j^b and v_j^c $(b < c)$ of the same writer, $\mathcal{W}_j$, such that $v_j^b \in B_u$ while $v_j^c \in \hat{B}_u$. Since $B_u \simeq \hat{B}_u$, it holds that $v_j^b \simeq v_j^c$, and in particular

$v_j^b.address = v_j^c.address$. A writer does not use the same address twice in a row, hence, $(b+1) < c$. Therefore, $r_u[j] \to w_j^{b+1} \to r_j^c[1] \to w_j^c$, which implies that L_j^c is linearized after c_s. If $v_j^c \not\equiv v_\ell^d$ then $\hat{G}_u$ has a path from v_j^c to v_ℓ^d, because v_j^c is in $\hat{B}_u$, and v_ℓ^d is last in $\hat{B}_u$. Hence, by Theorem 11 (b), $L_j^c \Rightarrow L_\ell^d$ and L_ℓ^d is linearized after c_s.

Case 2.2: $B_u \equiv \hat{B}_u$.

First, we show that B_u is a path in $H^{r_u[w]}$. To prove this, we assume that (v_j^b, v_k^c) is an arbitrary edge in B_u and show that (v_j^b, v_k^c) is an edge of $H^{r_u[w]}$. By Lemma 5, there exists an integer r, $r \geqslant 0$, such that (v_j^{b+r}, v_k^c) is an edge in $H^{w_k^c}$, where v_j^{b+r} is the node read in $\tilde{r}_k^c[j]$. Since v_k^c is a node of B_u, we conclude that $w_k^c \to r_u[w]$ and since no edge is ever deleted from the history graph, we conclude that (v_j^{b+r}, v_k^c) is an edge in $H^{r_u[w]}$. Since v_j^b is a node of B_u, and since $B_u \equiv \hat{B}_u$, we conclude that $\hat{r}_u[j] \to w_j^{b+1}$, thus v_j^b is the current node of $\mathcal{W}_j$ at least until $\hat{r}_u[j]$. Since $w_k^c \to r_u[w]$, we conclude that $\tilde{r}_k^c[j] \to w_k^c \to r_u[w] \to \hat{r}_u[j] \to w_j^{b+1}$ and since $r \geqslant 0$, we conclude that $r = 0$ and (v_j^b, v_k^c) is an edge of $H^{r_u[w]}$. The proof follows.

The assumptions $B_u \simeq \overleftarrow{B}_u$ (Case 2) and $B_u \equiv \hat{B}_u$ (Case 2.2) imply that $B_u \equiv \overleftarrow{B}_u \equiv \hat{B}_u$. Since $B_u \equiv B_H^{r_u[w]}$ (which is proven in Claim 2), we get that v_ℓ^d is last in $B_H^{r_u[w]}$, which implies that v_ℓ^d is lasting and that it is linearized at w_ℓ^d. Therefore, v_ℓ^d is either last in $B_H^{c_s}$ or it is linearized after c_s.

Claim 2. *Under the assumptions of Case* 2.2, *it holds that* $B_u \equiv B_H^{r_u[w]}$.

Proof of claim. Assume by way of contradiction that $B_u \not\equiv B_H^{r_u[w]}$. Since no edge is ever deleted from the history graph, it follows that for every configuration c_t, after $r_u[w]$, either $B_u \subset B_H^{c_t}$, or there exists some edge in H^{c_t} that excludes a suffix of B_u from $B_H^{c_t}$. Let $S = c_{\underset{t_w}{\leftarrow}}, c_{\underset{t_{w-1}}{\leftarrow}}, \ldots, c_{\underset{t_1}{\leftarrow}}$ be the sequence of result configurations of actions $\overleftarrow{r}_u[w]$, $\overleftarrow{r}_u[w-1], \ldots, \overleftarrow{r}_u[1]$, respectively. Define the function $EX(c_t)$ on S as follows: The value of $EX(c_t)$ is either w, if $B_u \subset B_H^{c_t}$ or it is k where k is the *id* of the node in $B_H^{c_t}$ whose edge excludes a suffix of B_u from $B_H^{c_t}$. For every $c_{\underset{t_i}{\leftarrow}}$, $1 \leqslant i \leqslant w$, $1 \leqslant EX(c_{\underset{t_i}{\leftarrow}}) \leqslant w$. Since no edge is ever deleted from B_H, the sequence of values of EX, $EX(c_{\underset{t_w}{\leftarrow}}), EX(c_{\underset{t_{w-1}}{\leftarrow}}), \ldots, EX(c_{\underset{t_1}{\leftarrow}})$ is non-increasing.

Now, we use the function EX to reach a contradiction by showing that under these assumptions $B_u \not\equiv \overleftarrow{B}_u$. Since $\overleftarrow{G}_u$ is collected in reverse order, from w to 1, and EX is a non-increasing integer function into the interval $[1, w]$, there exists an integer k, $1 \leqslant k \leqslant w$ such that $c_{\underset{t_k}{\leftarrow}} \in S$ and $EX(c_{\underset{t_k}{\leftarrow}}) = k$. Let (v_j^b, v_k^c) be the edge of $B_H^{c_{\underset{t_k}{\leftarrow}}}$ excluding a suffix of B_u from $B_H^{c_{\underset{t_k}{\leftarrow}}}$. By this definition, $v_j^b \in B_u$; since $B_u \equiv \overleftarrow{B}_u$, $v_j^b \in \overleftarrow{B}_u$, hence $v_j^b \in \overleftarrow{G}_u$. Since v_k^c belongs to $B_H^{c_{\underset{t_k}{\leftarrow}}}$, Lemma 6 implies that it is the current node of $\mathcal{W}_k$ at $c_{\underset{t_k}{\leftarrow}}$ and therefore v_k^c is a node in $\overleftarrow{G}_u$. Since $v_k^c \in \overleftarrow{G}_u$ and $v_j^b \in \overleftarrow{G}_u$, we get that (v_j^b, v_k^c) is an edge in $\overleftarrow{G}_u$ which excludes a suffix of B_u from $\overleftarrow{B}_u$, a contradiction to the assumption that $B_u \equiv \overleftarrow{B}_u$. □

Case 3: $B_u \not\simeq \hat{B}_u$ **and** $B_u \subset \hat{G}_u$

Let v_i^a be the last node of $\hat{B}_u$. According to the protocol S_u returns v_i^a. Since $B_u \subset \hat{G}_u$ and $B_u \not\simeq \hat{B}_u$, $\hat{B}_u$ has a suffix whose nodes are not in B_u. Let $v_j^c \in \hat{B}_u$ be the minimal node in that suffix and let v_j^b be the node of $\mathcal{W}_j$ in G_u. By this definition, $v_j^b \not\simeq v_j^c$. By Claim 3 below, L_j^c is linearized after c_s. If $L_j^c \equiv L_i^a$, we are done. Otherwise, $j < i$ and $\hat{B}_u$ has a path from v_j^c to v_i^a. By Theorem 11 (b), $L_j^c \Rightarrow L_i^a$.

Claim 3. *Under the assumptions of Case* 3, L_j^c *is linearized after* c_s.

Proof of claim. Since $v_j^b \not\simeq v_j^c$, clearly $r_u[j] \to w_j^c$ and therefore w_j^c occurs after c_s. If L_j^c is lasting then it is linearized at w_j^c and the proof follows. If L_j^c is transient then let v_k^d be the last node of $B_H^{w_j^c}$, by which L_j^c is linearized. If L_k^d is linearized after c_s then L_j^c is also linearized after c_s and we are done. We continue the proof assuming that L_k^d is linearized before c_s, that is, $w_k^d \to r_u[1]$. By this assumption, there are no lasting write actions that are linearized within the interval starting at $r_u[1]$ and ending at w_j^c. In this case, Lemma 9 implies that $B_H^{w_j^c}/(j+1) \equiv B_u/(j+1)$.

Consider (v_ℓ^e, v_j^c), the edge of v_j^c in $\hat{B}_u$. By Lemma 5, there exists an integer $r, r \geqslant 0$, such that (v_ℓ^{e+r}, v_j^c) is an edge in $H^{w_j^c}$. By Theorem 11(a), L_j^c is linearized after L_ℓ^{e+r}. Now, we show that $r > 0$: Assume by way of contradiction that $r = 0$, that is (v_ℓ^e, v_j^c) is an edge in $H^{w_j^c}$. By the definition of v_j^c, $v_\ell^e \in B_u$. Since $B_u/(j+1) \equiv B_H^{w_j^c}/(j+1)$, we get that $v_\ell^e \in B_H^{w_j^c}/(j+1)$. Since L_j^c is transient, let (v_ℓ^e, v_p^f) be the edge excluding (v_ℓ^e, v_j^c) from $B_H^{w_j^c}/(j+1)$. Since $B_u/(j+1) \equiv B_H^{w_j^c}/(j+1)$ the edge (v_ℓ^e, v_p^f) belongs also to $B_u/(j+1)$. Since $B_u \subset \hat{G}_u$, (v_ℓ^e, v_p^f) is an edge in $\hat{G}_u$, and it excludes (v_ℓ^e, v_j^c) from $\hat{B}_u$, a contradiction to the definition of v_j^c.

The proof is completed by showing that if $r > 0$ then L_ℓ^{e+r} is linearized after c_s, and therefore L_j^c is also linearized after c_s. If $r > 0$ then $r > 1$ because a writer never uses the same address twice in a row. Since v_ℓ^e is the current node read at $\hat{r}_u[\ell]$, $\hat{r}_u[\ell] \to w_\ell^{e+1}$. Since $w_\ell^{e+1} \to r_\ell^{e+r}[1]$ we get that $\hat{r}_u[\ell] \to r_\ell^{e+r}[1]$ which implies that L_ℓ^{e+r} is linearized after c_s. Therefore, L_j^c is also linearized after c_s. □

Case 4: $B_u \not\simeq \hat{B}_u$ **and** $B_u \not\subset \hat{G}_u$

Let i be the smallest *id* such that v_i^a is in B_u, $v_i^b \in \hat{G}_u$ and $v_i^a \not\simeq v_i^b$. According to the reader protocol, S_u returns v_i^b. Under this assumption, we show that L_i^b is linearized after c_s. Since $v_i^b \not\simeq v_i^a$, $w_i^a \to r_u[i] \to w_i^b \to \hat{r}_u[i]$, in particular $r_u[1] \to w_i^b$. If L_i^b is lasting, then it is linearized after c_s. Hence, for the rest of Case 4, we assume that L_i^b is transient. In the sequel, we show that there exists a lasting write action L_j^c which is linearized after c_s, and L_i^b is either linearized by L_j^c or it is linearized after L_j^c. Consider the following cases:

Case 4.1: $v_i^a \in B_H^{r_u[i]}$.

Since L_i^b is transient, $B_H^{w_i^b}/i \not\equiv B_i^b/i$, hence Lemma 9 (with $j = i-1$) implies that there exists a lasting write action L_j^c, such that $j < i$ and $r_i^b[j] \to w_j^c \to w_i^b$. Since $w_i^a \to r_i^b[j]$ we get that $w_i^a \to w_j^c \to w_i^b$.

Now, we use L_j^c to show that L_i^b is linearized after c_s. Since L_i^b is transient, it is linearized by the last node of $B_H^{w_i^b}$. Since $w_j^c \to w_i^b$, the last node of $B_H^{w_i^b}$ is either v_j^c or the node of another logical write action which is linearized after L_j^c.

The proof is completed by showing that L_j^c is linearized after $r_u[i]$, which implies that L_i^b is also linearized after $r_u[i]$ and therefore after c_s. Since L_j^c is lasting, it is linearized at w_j^c. Therefore, it is enough to show that $r_u[i] \to w_j^c$. Note that since $w_i^a \to w_j^c$, $v_i^a \in H^{w_j^c}$. However, since v_j^c is last in $B_H^{w_j^c}$ and $i > j$, Lemma 4(a) implies that $v_i^a \notin B_H^{w_j^c}$. Therefore, Lemma 1 implies that for any configuration c_t after w_j^c, $v_i^a \notin B_H^{c_t}$. By the assumption of this case $v_i^a \in B_H^{r_u[i]}$, therefore $r_u[i] \to w_j^c$.

Case 4.2: $v_i^a \notin B_H^{r_u[i]}$.

By the definition of i in Case 4, $v_i^a \in B_u$. On the other hand, by Case 4.2, $v_i^a \notin B_H^{r_u[i]}$, therefore $B_H^{r_u[i]}/(i+1) \not\equiv B_u/(i+1)$. By Lemma 9, there exists a lasting write action L_j^c, $j \leqslant i$, such that $r_u[j] \to w_j^c \to r_u[i]$, hence $j < i$. Since L_j^c is lasting, it is linearized at w_j^c. Since $w_j^c \to r_u[i] \to w_i^b$, we get $w_j^c \to w_i^b$, and therefore L_i^b is linearized either by L_j^c or later. Since w_j^c occurs after c_s, L_i^b is linearized after c_s too. □

5. Multi-writer registers using single-reader registers

5.1. Description

In this section, we present an implementation whose physical registers are atomic, $(1,1)$-registers. In this implementation, readers and writers use the same protocol which is obtained by modifying the writer protocol of the $(1,n)$ implementation. The *id*s of the readers are larger than the *id*s of the writers and the reader protocol contains an extra *return* statement in which the read value is returned. Communication is two sided; processor P_i communicates with processor P_j, $i \neq j$, by writing into a $(1,1)$ atomic register denoted by $REG_{i,j}$ from which P_j reads. A physical write action executed by P_i to $REG_{i,j}$ is denoted by $w_i[j]$; while $r_j[i]$ denotes a physical read action by P_j from $REG_{i,j}$. Logical action number a of P_i, where P_i is either a writer or a reader, is denoted by L_i^a. Since we use $(1,1)$ registers, some single physical actions of the $(1,n)$ implementation are replaced by n physical actions (for example w_i^a is replaced by $w_i^a[1] \cdots w_i^a[n]$). Each register contains five successive nodes called *new*, *current*, *previous*, *old* and *ancient*. Immediately after the occurrence of $w_i^a[j]$ the *current* node in $REG_{i,j}$ holds v_i^a, the node computed by L_i^a, while *previous*, *old* and *ancient* hold v_i^{a-1}, v_i^{a-2} and v_i^{a-3}, respectively.

The use of $(1,1)$ registers influences the protocols' design in two ways. On one hand, information propagation is not atomic any more: A processor that wishes to pass some information to all other processors should write n times whereas in the $(1,n)$ implementation a single atomic write would suffice. Consequently, information is passed to the processors *gradually* and not at once. On the other hand, since $(1,1)$ registers are used, each pair of processors can use a constant number of additional bits on top of those used for node encoding without increasing the $O(\log n)$ space complexity.

Now we discuss the influence of non-atomic information propagation on the protocol: Consider the situation when node v_j^b joins the system gradually during physical actions $w_j^b[1], \ldots, w_j^b[n]$. In these circumstances, the following problem may arise: let L_k^c be some action whose tail node is v_j^b, where $j < k$. Assume that L_k^c is terminated before $w_j^b[i], k < i$, occurs. If at this point, $P_\ell, \ell < i$, reads both $REG_{j,\ell}$ and $REG_{k,\ell}$, while executing L_ℓ^d, then nodes v_j^b and v_k^c belong to G_ℓ^d and (v_j^b, v_k^c) is an edge G_ℓ^d. On the other hand, if at this point, P_m, $m \geqslant i$, reads $REG_{j,m}$ and $REG_{k,m}$, while executing L_m^a, then, since $w_j^b[m]$ has not occurred yet, v_j^b is not a node of G_m^a and (v_j^b, v_k^c) is not included in G_m^a. This may cause a situation in which one reader returns one value while another reader returns a second value and atomicity might be violated. To prevent this problem the following adjustments are made:

(1) The protocol is augmented with the *inform* stage in which P_i informs all processors of its *new* node.
(2) During the invocation of *collect* in action L_i^a, all *new* nodes are scanned. Whenever P_i sees that the *new* node in $REG_{j,i}$, v_j^b, is the tail node of some node v_k^c, in G_i^a then v_j^b is added to G_i^a as well.

The code of (1,1) version of procedure *collect*, for P_i, appears in Fig. 11. In this implementation, the code depends on the *id* of P_i. Assume G_i^a is collected: For every processor P_j, G_i^a contains nodes *current*, *previous* and *old* read from $REG_{j,i}$. When G_i^a contains some node v_k^c whose tail node is the *new* node of P_j, v_j^b, (that is , v_j^b is the *new* field in $REG_{j,i}$) where $i > k > j$, v_j^b is added to G_i^a as well. The set *AD* contains the *tail_address* of all five nodes read from each register.

The gradual departure of nodes from the system raises a similar problem: Let E be an execution in which v_j^b is the tail node of v_i^a $(j < i)$. In actions $w_i^{a+3}[k], k = 1, 2, \ldots, n$, v_i^a is moved from the *old* fields into the *ancient* fields of $REG_{i,k}$. Consider the situation when actions $w_i^{a+3}[k], k = 1, 2, \ldots, \ell$, $\ell < n$ were executed, but actions $w_i^{a+3}[k], k = \ell + 1, \ell + 2, \ldots, n$, were not carried out yet. If, at this point, P_j uses $v_j^b.address$ as an address for some new node v_j^{b+r} $(r > 0)$ where L_j^{b+r} starts after L_i^a terminates, the edge (v_j^{b+r}, v_i^a) might be collected by some P_m, $\ell < m \leqslant n$. Since in E, $L_i^a \Rightarrow L_j^{b+r}$, this edge violates the precedence relation. To prevent this problem we add the *ancient* node to the registers of

```
Procedure collect(G, AD)
        V := {v_0^0}
        AD:=φ
        for j := 1 to n do
            reg_j := read(REG_{j,i})                                   r_i^a[j]
            (new, current, previous, old, ancient) := reg_j
            V := V ∪ {current, previous, old}
            AD := AD∪{new.tail_address, current.tail_address, previous.tail_address,
                      old.tail_address, ancient.tail_address}
        for j := i − 2 downto 1 do
            if ∃k, j < k < i, such that reg_j.new is the tail node of some node v_k^c ∈ V
              then V := V ∪ {reg_j.new}
        G := IND(V)
```

Fig. 11. Procedure *collect* for P_i in the (1,1) implementation.

the processors. The *ancient* node is never used as a tail node for new nodes; its role is to provide a window of time during which the *old* node leaves the system while reuse of its address is delayed.

Now, we present the notion of *enclosed actions*: Intuitively, action L_j^b is enclosed within L_i^a, if its execution interval [$r_j^b[1], w_j^b[n]$] is contained in the execution interval of L_i^a, [$r_i^a[1], w_i^a[n]$]. The *hand-shake* mechanism is a fairly standard distributed protocol which enables P_i to sometimes detect enclosed actions by processors with smaller *id*s. Whenever some enclosed actions are detected, the node of one of them is chosen as *can_tail*. In the next subsection, *enclosed* actions and *enclosed-free* actions are defined. The required properties of the hand-shake mechanism are stated in Lemma 13. The hand-shake mechanism is presented and Lemma 13 is proved in Appendix A.

The code for the protocol, without the details of the hand-shake mechanism, appears in Fig. 12. Now we describe the protocol assuming L_i^a is executed: Execution of L_i^a starts with an invocation of *collect* in which G_i^a is collected. The physical actions of *collect* are $r_i^a[1] \cdots r_i^a[n]$. Following that,

```
begin
    collect(G_i, AD)                                                    r_i^a[1] ... r_i^a[n]
    AD := AD ∪ {current.address, previous.address,
                        old.address, ancient.address}
    if L_i is enclosed-free then
        can_tail := last node in B_i/i
    else
        can_tail := node with maximal id enclosed in L_i^a
    endif
    cid := can_tail.id
    new.address := min(AD − AD)
    new.tail_id := cid
    new.tail_address := can_tail.address
    REG_{i,cid} := write (new, current, previous, old, ancient)          p_i^a[cid]
    nreg := read(REG_{cid,i})                                           r̃_i^a[cid]
    Case                    /*L_i^a connects*/
        [can_tail ≃ nreg.current]:      Ret_node := nreg.current
        [can_tail ≃ nreg.previous]:     Ret_node := nreg.previous
        [can_tail ≃ nreg.old]:          Ret_node := nreg.old
    else                    /*L_i^a loops*/
        new.tail_id := i
        new.tail_address := new.address
        Ret_node := can_tail
    endif
    for j := 1 to n
        REG_{i,j} := write (new, current, previous, old, ancient)        i_i^a[1] ... i_i^a[n]
    endfor
    current, previous, old, ancient := new, current, previous, old
    for j := 1 to n
        REG_{i,j} := write (new, current, previous, old, ancient)        w_i^a[1] ... w_i^a[n]
    endfor
    if you are a reader (i > w) then return(Ret_node.value)
end
```

Fig. 12. The protocol for P_i in the (1, 1) implementation.

the frontal branch of G_i^a, B_i^a, is computed. Computation of B_i^a is done just like before, where relation *locally precedes* is defined so that each of *old*, *previous*, *current* and *new* locally precedes its successors on the list. After that, P_i computes its tentative *new* node, whose tail node is denoted by *can_tail*, as follows: If an enclosed node is detected then *can_tail* is the enclosed node with maximal *id*. Otherwise, *can_tail* is the last node of B_i^a/i. As before, *cid* denotes the *id* of (the owner of) *can_tail*. The address of *new* is the minimal address not included in the set *AD* which includes components *tail_address* of all nodes read during *collect*. In addition, it is ensured that the address of *new* is not equal to the last four addresses used by P_i, thus, the addresses of every five consecutive nodes are distinct and if $v_i^a \simeq v_i^{a+r}$ and $r > 0$, then $r > 4$.

During execution of L_i^a, P_i computes a node called *Ret_node* as follows: If L_i^a connects then *Ret_node* is chosen as the tail node of v_i^a, if L_i^a loops then *Ret_node* is equal to node *can_tail* computed during L_i^a. Node *Ret_node* is used only if P_i is a reader, in this case the value corresponding to *Ret_node* is returned by L_i^a and it becomes the value corresponding to v_i^a. This is the only place in the implementation in which a value corresponding to one node is copied to another node.

In the next step, P_i declares the *new* node, to P_{cid} exclusively, in action $p_i^a[cid]$. Then, in action $\tilde{r}_i^a[cid]$, P_i rereads $REG_{cid,i}$. If v_{cid} is $\simeq$ to either *current*, or *previous*, or *old* then L_i^a connects—the tentative choice of *tail* is committed; otherwise (L_i^a loops), the tail node of v_i^a is v_i^a itself. Thus, after this stage, the choice of node *new* is finalized. After choosing its final new node P_i executes the *inform* stage in which it informs all other processors about the new value by writing it into component *new* of all its registers in physical actions $i_i^a[1] \cdots i_i^a[n]$. Logical action L_i^a is concluded with physical actions $w_i^a[1] \cdots w_i^a[n]$ in which v_i^a is written in the *current* field of all registers of P_i one after the other.

To complete definition of the protocol, we need to specify initial values for all data structures. Similar to the $(1, n)$ implementation, all nodes in all registers are initialized to hold the initial node, v_i^0, where $v_i^0.address = 0$, $v_i^0.tail\_id = i$ and $v_i^0.tail\_address = 0$. That is, at the initial configuration all edges of the current graph are self-loops, the initial frontal branch contains only the root node v_0^0 and the initial logical value is the value corresponding to v_0^0, which can be chosen freely from the set of all permitted values. The initial values for the bits implementing the hand-shake mechanism are presented in Appendix A.

Since there are five nodes in each register, and each node contains three fields each of which is of size $\log n + O(1)$ bits, the space complexity of the $(1, 1)$ implementation is $\Theta(\log n)$. The time complexity, including the implementation of the hand-shake mechanism, is $5n + O(1)$.

5.2. Linearization of the logical actions

Similar to the previous section, we fix some arbitrary execution of the system, $E = c_0, d_1, c_1, \ldots$, and all definitions and proofs are made with respect to E.

Definition 10. The *tail node* of v_i^a is defined as follows:

(1) If L_i^a connects then the tail node of v_i^a is *Ret_node*, read in $\tilde{r}_i^a[cid]$, which is $\simeq$ to *can_tail*.
(2) If L_i^a loops then the tail node of v_i^a is v_i^a itself.

In this implementation, once more a *history graph*, H, is used to linearize the logical actions. For every processor, P_i, writer or reader, each node v_i^a is included in H. The configuration at which v_i^a

joins the history graph, which is called the *joining configuration* of v_i^a, should satisfy the following requirements:

(1) The joining configuration of v_i^a lies within the interval $[w_i^a[i+1],\ w_i^a[n]]$.
(2) If (v_j^b, v_i^a), $j < i$, is an edge in H then the joining configuration of v_i^a is not earlier than the joining configuration of v_j^b.

These requirements are fulfilled by the following definition.

Definition 11. Let L_i^a be some logical action and let DES_i^a be the set of actions whose nodes are descendents of v_i^a in H, including v_i^a itself. Let L_j^b be the first logical action in DES_i^a to complete its execution. The *joining configuration* of v_i^a (and all nodes on the directed path from v_i^a to v_j^b) is the result configuration of $w_j^b[n]$. In case $v_i^a \neq v_j^b$, we say that v_i^a joins the history graph by v_j^b.

Definition 12. Let E be an execution of the system. The *History graph* of E, H, is defined as follows:

(1) H^0 is the *History graph* before the execution begins. It contains only the *root* node—v_0^0.
(2) Let c_{t_j} be the result configuration of $w_j^b[n]$. Assume that c_{t_j} is the joining configuration of a set of nodes PTH_i^j, $i \leqslant j$. In this case we define $H^{c_{t_j}}$ as the graph obtained from $H^{c_{t_j-1}}$ by adding all nodes of PTH_i^j. For each node of PTH_i^j, v_k^c, $H^{c_{t_j}}$ contains an edge emanating from the tail node of v_k^c and incoming to v_k^c. If c_{t_j} is not a joining configuration of any node then $H^{c_{t_j}} \equiv H^{c_{t_j-1}}$.

Definition 13. Let c_{t_i} be the joining configuration of v_i^a. Action L_i^a is *lasting* if v_i^a belongs to $B_H^{c_{t_i}}$. An action which is not lasting is *transient*.
Note. Since in this implementation, configuration c_{t_i} may be the joining configuration of *several* nodes. A node of a lasting action is only required to belong to $B_H^{c_{t_i}}$ (and not necessarily to be last in $B_H^{c_{t_i}}$).

Now, we define enclosed actions: Intuitively, action L_j^b is enclosed within L_i^a, if its execution interval $[\ r_j^b[1], w_j^b[n]\ ]$ is contained in the execution interval of L_i^a, $[\ r_i^a[1], w_i^a[n]\ ]$. Since if $i \neq n$, P_i cannot determine the occurrence time of $w_j^b[n]$ we define L_j^b as enclosed in L_i^a if the interval $[\ r_j^b[1], w_j^b[i]\ ]$ is contained in the interval $[\ r_i^a[1], w_i^a[n]\ ]$. Even under this permissive definition, asynchronicity makes it impossible to detect every enclosed action. The maximum one can hope for is that if P_j executes "enough" enclosed actions during L_i^a then one of them is detected. In the following definition we use a slightly abusive language and define an action as enclosed-free if no enclosed action is *detected*, that is, an enclosed-free action may actually have some (undetected) enclosed actions. Detection of enclosed actions is done by use of the hand-shake mechanism. Description of the mechanism and a formal proof of its properties appear in Appendix A.

Definition 14. L_j^b and v_j^b are *enclosed* within L_i^a, $i > j$, if the execution of L_j^b begins after the execution of L_i^a, and v_j^b is the *current* node in $REG_{j,i}$, collected during L_i^a. Action L_i^a is *enclosed-free* if it does not *detect* any enclosed action.

Now, we define the linearization point for the logical actions. Logical write actions are linearized independently of the logical read actions according to the following definition.

Definition 15. Let L_i^a be a logical write action whose joining configuration is c_{t_i}.

(1) If L_i^a is last in $B_H^{c_{t_i}}/(w+1)$ then L_i^a is linearized at c_{t_i}.
(2) If v_j^b is the last node of $B_H^{c_{t_i}}/(w+1)$ then L_i^a is linearized by L_j^b, that is, before L_j^b and after any other physical action that precedes the joining configuration of v_j^b. In case several logical write actions are linearized by the same logical action they are linearized in ascending order of their *id*s.

Read actions are linearized according to the nodes they return. The linearization point of a logical read action L_i^a is defined as follows.

Definition 16. Let L_i^a be a logical read action whose *Ret_node* is v_j^b. If L_j^b is linearized before the beginning of the execution interval of L_i^a, then L_i^a is linearized at the beginning of its execution interval. Otherwise, L_i^a is linearized by L_j^b, that is, after L_j^b and before any physical action which occurs after L_j^b, or before any logical action which is linearized, after L_j^b. In case several logical read actions are linearized by the same logical action they are linearized in ascending order of their *id*s.

5.3. Correctness proof

We begin the proof with several auxiliary lemmas which are used in the proof of Theorem 19 in which it is proved that every logical action is linearized inside its execution interval. At the end of the proof, in Theorem 22, we show that every logical read action returns the value written by the last logical write action linearized before it, hence, the implementation is correct.

The first lemma deals with enclosed actions and with the properties of their detection mechanism. A description of the mechanism and a proof for the lemma appear in Appendix A.

Lemma 13. *Let v_j^b be the last node of P_j whose joining configuration is before $r_i^a[1]$. There exists a detection mechanism that requires three bits for each pair of processors such that if, for some $r \geqslant 0$, v_j^{b+3+r} is the current node in $REG_{j,i}$ at $r_i^a[j]$, then it is detected as enclosed.*

Lemma 14. *Let (v_j^b, v_i^a), $j < i$, is an edge in G_k^c and let c_{t_j} and c_{t_i} be the joining configuration of v_j^b and v_i^a, respectively. Under these conditions the following hold*:

(1) *There exists some integer r, $r \geqslant 0$ such that (v_j^{b+r}, v_i^a) is an edge in $H^{c_{t_i}}$.*
(2) *Configuration c_{t_j} is not later than c_{t_i}.*

Proof of 1. By the protocol, if (v_j^b, v_i^a) is an edge of G_k^c, where $j < i$, then L_i^a connects, and its tail node is some node of P_j, v_j^{b+r}. To prove the lemma we show that $r \geqslant 0$.

Since (v_j^b, v_i^a) is an edge of G_k^c and (v_j^{b+r}, v_i^a) is an edge of H, we conclude that $v_j^b.address = v_j^{b+r}.address$. In the next claim we assume by way of contradiction that $r < 0$ and prove that

$p_i^a[j] \to r_j^b[i] \to w_i^{a+4}[j]$. Since v_i^a is one of the five nodes in $REG_{i,j}$ throughout the interval $[\, p_i^a[j]\,,\ w_i^{a+4}[j]\,)$, we conclude that $v_i^a.tail\_address \in AD_j^b$, where AD_j^b is the set AD returned by procedure *collect* invoked during L_j^b. By the protocol, $v_j^b.address \neq v_i^a.tail\_address$. On the other hand, the fact that (v_j^b, v_i^a) is an edge of G_k^c implies that $v_j^b.address = v_i^a.tail\_address$, a contradiction.

Claim 4. *Under the conditions of the lemma, if* $r < 0$ *then* $p_i^a[j] \to r_j^b[i] \to w_i^{a+4}[j]$.

Proof of claim. First, we show that $p_i^a[j] \to r_j^b[i]$. Since the addresses of every five consecutive nodes are distinct and $v_j^{b+r}.address = v_j^b.address$, the fact that $r < 0$ implies that $r < -4$. Nodes *old*, *previous* and *current* in $REG_{j,i}$ at $w_j^{b-2}[i]$ are v_j^{b-4}, v_j^{b-3} and v_j^{b-2}, respectively. Since $r < -4$, v_j^{b+r} is neither *old* nor *previous* nor *current* in $REG_{j,i}$ at $w_j^{b-2}[i]$. Since L_i^a connects, v_j^{b+r} is one of these nodes in $REG_{j,i}$ at $\tilde{r}_i^a[j]$, hence $\tilde{r}_i^a[j] \to w_j^{b-2}[i]$. By the protocol, $p_i^a[j] \to \tilde{r}_i^a[j]$ and $w_j^{b-2}[i] \to r_j^b[i]$, hence $p_i^a[j] \to r_j^b[i]$.

The fact that $r_j^b[i] \to w_i^{a+4}[j]$ is proved as follows:

(1) $r_j^b[i] \to i_j^b[k]$ (according to the protocol)
(2) $i_j^b[k] \to r_k^c[j]$ (since $v_j^b \in G_k^c$)
(3) $r_k^c[j] \to r_k^c[i]$ (since $i > j$ and G_k^c is collected in ascending order)
(4) $r_k^c[i] \to w_i^{a+3}[k]$ (since $v_i^a \in G_k^c$)
(5) $w_i^{a+3}[k] \to w_i^{a+4}[j]$ (P_i works in sequential manner)

Since the right-hand side of each relation is the left-hand side of the next one, we get $r_j^b[i] \to w_i^{a+4}[j]$. In conclusion, $p_i^a[j] \to r_j^b[i] \to w_i^{a+4}[j]$. The claim follows. □

Proof of 2. Let $c_{t_{j+r}}$ be the joining configuration of v_j^{b+r}. By Definition 11, $c_{t_{j+r}}$ is not later than c_{t_i}. If $r > 0$, the protocol implies that $c_{t_{j+r}}$ is later than c_{t_j}. □

Lemma 15. *Let* c_{t_i} *be the joining configuration of* L_i^a *and let* v_j^b *be node can_tail computed by* L_i^a. *Under these conditions* $v_j^b \in H^{c_{t_i}}$.

Proof. Let v_j^b be node *can_tail* in L_i^a. First we show that v_j^b is not the *new* node in $REG_{j,i}$ at $r_i^a[j]$. If L_i^a is enclosed-free then v_j^b is the last node in B_i^a/i. Since a *new* node of P_ℓ is added to G_i^a only if it is the tail node of another node v_k^c, where $\ell < k < i$, we conclude that v_j^b is not *new* in $REG_{j,i}$. If L_i^a has an enclosed action then, by Definition 14, v_j^b is the *current* node of P_j, where j is largest *id* of detected enclosed action.

Now, we continue the proof, assuming that v_j^b is not *new* in $REG_{j,i}$. If L_i^a loops, then v_j^b is not among *current*, *previous* or *old* read from $REG_{j,i}$ in $\tilde{r}_i^a[j]$. Obviously $w_j^b[n] \to r_i^a[j] \to w_i^a[n]$ and we are done. Assume that L_i^a connects. In this case $v_j^b \simeq$ one of *current*, *previous* or *old* in $REG_{j,i}$ at $\tilde{r}_i^a[j]$. Assume $v_j^b \simeq v_j^c$, where v_j^c is one of these three fields. If $v_j^b \not\equiv v_j^c$, we conclude that $w_j^b[n] \to i_j^c[j] \to \tilde{r}_i^a[j]$ and, once again, we are done. Assume now that $v_j^b \equiv v_j^c$. In this case, v_j^b is the tail node of v_i^a in H. By Definition 11, $v_j^b \in H^{c_{t_i}}$. □

Lemma 16. *Let L_k^c be an enclosed-free action and let c_{t_k} be the joining configuration of v_k^c. Every node of B_k^c/k belongs to $H^{c_{t_k}}$.*

Proof. Let the nodes of B_k^c/k be $v_{j_0}(=v_i^a), v_{j_1}, \ldots, v_{j_p}(=v_0^0)$, where v_i^a is *last* in B_k^c/k. We prove the lemma by induction on j_q.

Base:
We have to show that the last node of B_k^c/k, v_i^a, satisfies the lemma. Since L_k^c is enclosed-free, it holds that v_i^a is node *can_tail* computed during L_k^c. By Lemma 15, $v_i^a \in H^{c_{t_k}}$.

Induction step:
Assume that all nodes $v_{j_0}, \ldots, v_{j_q}$ belong to $H^{c_{t_k}}$. To complete the proof it suffices to show that $v_{j_{q+1}} \in H^{c_{t_k}}$. Assume that $(v_{j_{q+1}}, v_{j_q}) = (v_j^b, v_\ell^e)$ for some processors P_j and P_ℓ. By Lemma 14, there exists some r, $r \geqslant 0$, such that (v_j^{b+r}, v_ℓ^e) is an edge in $H^{c_{t_k}}$, that is, $v_j^{b+r} \in H^{c_{t_k}}$. If $r = 0$, we are done. If $r > 0$, then since $L_j^b \to L_j^{b+r}$, clearly $v_j^b \in H^{c_{t_k}}$ as well. □

Let L_i^a be a logical action. The *initial configuration* of L_i^a is the configuration that precedes the first physical action of L_i^a, that is the configuration that precedes $r_i^a[1]$.

Lemma 17. *Let c_{s_i} be the initial configuration of L_i^a and let c_{t_i} be the joining configuration of v_i^a.*

1. *If $B_H^{c_{s_i}}/i \equiv B_H^{c_{t_i}}/i$ then*
   (1.a) *L_i^a is enclosed-free.*
   (1.b) *$B_H^{c_{s_i}}/i$ is a subgraph of G_i^a.*
   (1.c) *L_i^a is lasting.*
2. *For any $c \geqslant c_{t_i}$, if $v_i^a \in B_H^c$ then $v_i^{a+1} \notin H^c$.*

Proof. By induction on i, the *id*s of the processors.

Base: $i = 1$

(1.a) Every logical action executed by P_1 is enclosed-free.
(1.b) For any configuration c, $B_H^c/1$ contains a single node, namely v_0^0.
(1.c) Every action of P_1 is lasting.
(2) Every node of P_1 excludes its previous node from the frontal branch of the history graph.

Step: Assume correctness for $j < i$. Now, we prove correctness for i.

Proof of (1.a). Assume by way of contradiction that L_i^a is not enclosed-free. By Definition 14, this means that L_i^a detects at least one enclosed action. Let L_j^b, $j < i$, be the action with maximal *id*, detected as enclosed in L_i^a. By Definition 14, the initial configuration of L_j^b, c_{s_j}, is after the initial configuration of L_i^a, c_{s_i}. By the protocol, v_j^b is node *can_tail* computed by L_i^a, hence, Lemma 15 implies that $v_j^b \in H^{c_{t_i}}$. We get that the interval $[c_{s_j}, c_{t_j}]$ is contained within the interval $[c_{s_i}, c_{t_i}]$. Since $B_H^{c_{s_i}}/i \equiv B_H^{c_{t_i}}/i$ we get that $B_H^{c_{s_j}}/j \equiv B_H^{c_{t_j}}/j$. By Induction Assumption (1.c), L_j^b is lasting. Hence, the frontal branch of

the history graph is modified at c_{t_j}, the joining configuration of v_j^b. Since $j < i$, $B_H^{c_{s_j}}/i \not\equiv B_H^{c_{t_j}}/i$, a contradiction.

Proof of (1.b). We have to show that every node of $B_H^{c_{s_i}}/i$ belongs to G_i^a: First, we show that every such node is read during the collection of G_i^a either as *new* or as *current* or as *previous*. Let v_j^b be some node in $B_H^{c_{s_i}}/i$. Clearly $i_j^b[i] \to c_{s_i} \to r_i^a[j]$. Since $B_H^{c_{s_j}}/i \equiv B_H^{c_{t_i}}/i$ we get that $v_j^b \in B_H^{c_{t_i}}/i$ which implies, by induction hypothesis (2), that $v_j^{b+1} \notin H^{c_{t_i}}$. Therefore, $w_j^{b+1}[n]$ occurs after c_{t_i} which implies that $i_j^b[i] \to r_i^a[j] \to w_j^{b+1}[n]$. Thus, v_j^b is either *new* or *current* or *previous* in $REG_{j,i}$ at $r_i^a[j]$. If v_j^b is either *current* or *previous* then it belongs to G_i^a independently of the rest of the nodes of G_i^a. If, however, v_j^b is *new* in $REG_{j,i}$ at $r_i^a[j]$ then it belongs to G_i^a only if it is the tail node of another node in G_i^a. Thus, to show that all the nodes of $B_H^{c_{s_i}}/i$ belong to G_i^a, it suffices to show that the last node of $B_H^{c_{s_i}}/i$ is in G_i^a.

Let v_ℓ^d be the last node of $B_H^{c_{s_i}}/i$. Now, we show that $w_\ell^d[i] \to r_i^a[j]$ which implies that v_ℓ^d is either *current* or *previous* in $REG_{j,i}$ at $r_i^a[j]$, hence, $v_\ell^d \in G_i^a$. If the joining configuration of v_ℓ^d is at $w_\ell^d[n]$, then $w_\ell^d[n] \to c_{s_i} \to r_i^a[j]$. Otherwise, v_ℓ^d joins the history graph by another node v_k^c whose tail node is v_ℓ^d, where $\ell < i \leqslant k$. Since $v_\ell^d \in B_H^{c_{s_i}}$ and v_ℓ^d joins the history graph by v_k^c, Definition 11 implies that $v_k^c \in H^{c_{s_i}}$. The proof is completed by the following relations:

1.a. $w_\ell^d[i] = w_\ell^d[k]$ (if $i = k$). **Or**
1.b. $w_\ell^d[i] \to w_\ell^d[k]$ (according to the protocol, assuming $i < k$).
2. $w_\ell^d[k] \to \tilde{r}_k^c[j]$ (since v_ℓ^d is the tail node of v_k^c).
3. $\tilde{r}_k^c[j] \to c_{s_i}$ (since $v_k^c \in H^{c_{s_i}}$).
4. $c_{s_i} \to r_i^a[j]$ (according to the protocol).

Which imply $w_\ell^d[i] \to r_i^a[j]$. Therefore, $v_\ell^d \in G_i^a$, and $B_H^{c_{s_i}}/i$ is a subgraph of G_i^a.

Proof of (1.c). Assume by way of contradiction that L_i^a is transient. Since $B_H^{c_{s_i}}/i \equiv B_H^{c_{t_i}}/i$, induction hypothesis (1.a) and (1.b) for i imply that L_i^a is enclosed-free and that $B_H^{c_{t_i}}/i$ is a subgraph of G_i^a. Since L_i^a is enclosed-free, Lemma 16 implies that B_i^a is a subgraph of $H^{c_{t_i}}$. Now, we show that L_i^a does not loop: We claim that B_i^a is a subgraph of $B_H^{c_{t_i}}$. Assume towards a contradiction that v_j^b is the last node in their common prefix and (v_j^b, v_k^c) is the first edge of B_i^a not in $B_H^{c_{t_i}}$. Since B_i^a is a subgraph of $H^{c_{t_i}}$, the edge (v_j^b, v_k^c) belongs to $H^{c_{t_i}}$. Since (v_j^b, v_k^c) is not an edge of $B_H^{c_{t_i}}$, there exists an edge of $B_H^{c_{t_i}}$, (v_j^b, v_ℓ^d), excluding (v_j^b, v_k^c) from $B_H^{c_{t_i}}$. Since $B_H^{c_{t_i}}/i$ is a subgraph of G_i^a, we conclude that (v_j^b, v_ℓ^d) is an edge of G_i^a and it excludes (v_j^b, v_k^c) from B_i^a, a contradiction, thus B_i^a is a subgraph of $B_H^{c_{t_i}}$. Let $v_j^b, j < i$, be node *can_tail* selected during L_i^a. Since B_i^a is a subgraph of $B_H^{c_{t_i}}$ and L_i^a is enclosed-free, v_j^b is a node of $B_H^{c_{t_i}}/i$. Since $\tilde{r}_i^a[j] \to c_{t_j}$, v_j^b is read either as *new*, or as *previous* or as *old* during $\tilde{r}_I^a[j]$, hence L_i^a does not loop.

Since L_i^a is enclosed-free, transient and does not loop, we get that $B_i^a/i \not\equiv B_H^{c_{t_i}}/i$. Let $v_j^b, j < i$, be the node with the maximal *id* in their common prefix. Node v_j^b is not last in B_i^a/i since $B_H^{c_{t_i}}/i$ is a subgraph of G_i^a and $B_i^a/i \not\equiv B_H^{c_{t_i}}/i$. Let (v_j^b, v_k^c), $k < i$, be the edge emanating from v_j^b in B_i^a/i. Now, we show that (v_j^b, v_k^c) is an edge in $H^{c_{t_i}}$: Since L_i^a is enclosed-free, Lemma

16 implies that $v_k^c \in H^{c_{t_i}}$. By Lemma 14, there exists some $r \geqslant 0$, such that (v_j^{b+r}, v_k^c) is an edge in $H^{c_{t_i}}$. Since $v_j^b \in B_H^{c_{t_i}}$, induction hypothesis (2) implies that $v_j^{b+1} \notin H^{c_{t_i}}$, that is $r = 0$ and (v_j^b, v_k^c) is an edge in $H^{c_{t_i}}$. By the maximality of v_j^b, $(v_j^b, v_k^c) \notin B_H^{c_{t_i}}/i$. Let (v_j^b, v_ℓ^d), $\ell < i$, be the edge excluding (v_j^b, v_k^c) from $B_H^{c_{t_i}}/i$. Since $B_H^{c_{t_i}}/i$ is a subgraph of G_i^a, $(v_j^b, v_\ell^d) \in G_i^a$, and it excludes (v_j^b, v_k^c) from B_i^a/i, a contradiction.

Proof of (2). Assume by way of contradiction that v_i^a does not satisfy the lemma, that is $v_i^a \in B_H^{c_{u_i}}$ where c_{u_i} is the joining configuration of v_i^{a+1}. Since $v_i^a \in B_H^{c_{u_i}}$, L_i^{a+1} is transient. We reach the required contradiction by showing that L_i^{a+1} is lasting. Since the joining configuration of v_i^a is before $r_i^{a+1}[1]$ and $v_i^a \in B_H^{c_{u_i}}$, there are no lasting actions with *id* smaller then i in the interval $[r_i^{a+1}[1],\ c_{u_i}]$. Hence, $B_H^{r_i^{a+1}[1]}/i \equiv B_H^{c_{u_i}}/i$ which implies, by (1.c), that L_i^{a+1} is lasting. □

Lemma 18. *If L_i^a is enclosed-free then $B_H^{r_i^a[1]}/i$ is a subgraph of G_i^a.*

Proof. In order to show that $B_H^{r_i^a[1]}/i$ is a subgraph of G_i^a, we show that every node of $B_H^{r_i^a[1]}/i$ belongs to G_i^a: Let v_j^b be an arbitrary node in $B_H^{r_i^a[1]}/i$. Since $v_j^b \in B_H^{r_i^a[1]}/i$, $i_j^b[i] \to r_i^a[1] \to r_i^a[j]$. By Lemma 17(2), v_j^b is the last node of P_j whose joining configuration is before $r_i^a[1]$. Since L_i^a is enclosed-free, Lemma 13 implies that the current node in $REG_{j,i}$ at $r_i^a[j]$ is v_j^{b+r}, $0 \leqslant r \leqslant 2$. Thus, v_j^b is read, in $r_i^a[j]$, during the collection of G_i^a either as *new* or as *current* or as *previous* or as *old* in $REG_{j,i}$. If v_j^b is either *current* or *previous* or *old* then it belongs to G_i^a independently of the rest of the nodes in G_i^a. Assume that v_j^b is *new* in $REG_{j,i}$ at $r_i^a[j]$. The proof that $v_j^b \in G_i^a$ is identical to the proof of Lemma 17(1.b). □

Theorem 19. *Every logical action is linearized within its execution interval.*

Proof. Since logical write and read actions are linearized differently, we prove the theorem separately for each type of logical action:

Proof for logical write actions. Let L_i^a be a logical write action whose initial configuration is c_{s_i}. By Definition 15, every logical write action is linearized no later than its joining configuration. Therefore, we only have to show that L_i^a is linearized after c_{s_i}. Let c_{t_i} be the joining configuration of v_i^a. If L_i^a is lasting then it is linearized either at c_{t_i} or just before it, and the theorem follows. Assume that L_i^a is transient. By Lemma 17(1.c), $B_H^{c_{s_i}}/i \not\equiv B_H^{c_{t_i}}/i$, hence, there exists a lasting logical write action L_j^b, $j < i$, whose joining configuration is after c_{s_i} and before c_{t_i}. Action L_i^a is either linearized by L_j^b or it is linearized by another lasting logical write action which is linearized after L_j^b and before c_{t_i}. The theorem follows.

Proof for logical read actions. Now, we prove by induction on i, the *id*s of the readers that every logical read action of P_i is linearized within its execution interval:

Base: $i = w + 1$

Let L_i^a be a logical write action of P_i and let c_{s_i} and c_{t_i} be the initial configuration of L_i^a and the joining configurations of v_i^a, respectively. By Definition 16, every logical read action is linearized after its initial configuration, therefore we only have to prove that L_i^a is linearized

no later than c_{t_i}. Let v_j^b, $j < i$, be node *ret_node* computed by L_i^a. Since $j < i$, P_j is a writer, hence, the first part of the proof implies that L_j^b is linearized within its execution interval. If L_i^a connects then by Definition 11, $v_j^b \in H^{c_{t_i}}$. Therefore, Definition 16 implies that L_i^a is linearized before c_{t_i}. If L_i^a loops, then obviously L_j^b is completed before c_{t_i}, so once again, L_i^a is linearized before c_{t_i}.

Induction step:

Assume that all logical actions of all readers whose *id*s are less then i are linearized within their execution interval and let L_i^a be an arbitrary action of P_i, whose *ret_node* is v_j^b. Action L_j^b is linearized within its execution interval, either by the first part of the theorem, if P_j is a writer, or by the induction hypothesis, if P_j is a reader. In both cases, the proof is identical to the proof of the base case. □

Theorem 20. *If (v_j^b, v_i^a), $j < i$, is an edge in G_m^e then $L_j^b \Rightarrow L_i^a$.*

Proof. Let c_{t_i} be the joining configuration of v_i^a. By Lemma 14, there exists some $r \geqslant 0$ such that (v_j^{b+r}, v_i^a) is an edge of $H^{c_{t_i}}$. We show that $L_j^{b+r} \Rightarrow L_i^a$. The theorem follows since if $r > 0$, $L_j^b \Rightarrow L_j^{b+r}$. Consider the following cases:

Case 1: $i \leqslant w$.

In this case L_j^{b+r} and L_i^a are both write actions. If L_i^a is lasting or if both L_i^a and L_j^{b+r} are transient then the proof is implied immediately by Definition 15. If L_j^{b+r} is lasting and L_i^a is transient then L_i^a joins the history by another lasting logical write action, L_k^c, that satisfies $L_j^{b+r} \Rightarrow L_k^c$. By Definition 15, $L_j^{b+r} \Rightarrow L_i^a$.

Case 2: $j \leqslant w < i$.

In this case the value written by L_j^{b+r} is returned by L_i^a. By Definition 16, $L_j^{b+r} \Rightarrow L_i^a$.

Case 3: $w \leqslant j$.

In this case, by Definition 16, L_i^a is linearized by L_j^b. □

Lemma 21. *Let $L_i^a, i > w$, be an enclosed-free read action and let v_j^b, $j \leqslant w$, be the last node of $B_i^a/w+1$. If $v_j^b \notin B_H^{r_i^a[1]}/w+1$ then L_j^b is linearized after $r_i^a[1]$.*

Proof. Since $v_j^b \in B_i^a/(w+1)$ but $v_j^b \notin B_H^{r_i^a[1]}/(w+1)$, $B_i^a/(w+1) \not\equiv B_H^{r_i^a[1]}/(w+1)$. Let v_k^c be the node with the maximal *id* in the common prefix. Since $v_j^b \in B_i^a/(w+1)$, v_k^c is not last in $B_i^a/(w+1)$. Let (v_k^c, v_ℓ^d) be the edge emanating from v_k^c in $B_i^a/(w+1)$. Let c_{t_ℓ} be the joining configuration of v_ℓ^d. By the next claim c_{t_ℓ} is after $r_i^a[1]$. If $L_\ell^d \equiv L_j^b$, we are done. If $L_\ell^d \not\equiv L_j^b$ then there is a path from v_ℓ^d to v_j^b in G_i^a (because v_j^b and v_ℓ^d are both of B_i^a/i and v_j^b is last in B_i^a/i). By Theorem 20, $L_\ell^d \Rightarrow L_j^b$, therefore L_j^b is linearized after $r_i^a[1]$.

Claim 5. *Under the conditions of the lemma, c_{t_ℓ} is after $r_i^a[1]$.*

Proof of claim. Assume by way of contradiction that c_{t_ℓ} is before $r_i^a[1]$, that is, $v_\ell^d \in H^{r_i^a[1]}$. By Lemma 14, there exists some $r, r \geqslant 0$, such that (v_k^{c+r}, v_ℓ^d) is an edge of $H^{r_i^a[1]}$. Since $v_k^c \in B_H^{r_i^a[1]}/(w+1)$, Lemma

17(2) implies that $v_k^{c+1} \notin H^{r_i^a[1]}/(w+1)$. Hence, $r = 0$ and (v_k^c, v_ℓ^d) is an edge of $H^{r_i^a[1]}$. Let (v_k^c, v_m^e), $m \leqslant \ell$, be the edge excluding (v_k^c, v_ℓ^d) from $B_H^{r_i^a[1]}$. Since L_i^a is enclosed-free, Lemma 18 implies that $B_H^{r_i^a[1]}/i$ is a subgraph of G_i^a. Hence, (v_k^c, v_m^e) is an edge of G_i^a and it excludes (v_k^c, v_ℓ^d) from B_i^a, a contradiction. Therefore, the joining configuration of v_ℓ^d, c_{t_ℓ} is after $r_i^a[1]$.

Now, we show that L_ℓ^d is linearized after $r_i^a[1]$. If $B_H^{r_i^a[1]}/\ell \not\equiv B_H^{c_{t_\ell}}/\ell$, then L_ℓ^d is linearized by the lasting write action that is linearized last before c_{t_ℓ}, that is after $r_i^a[1]$. Assume that $B_H^{r_i^a[1]}/\ell \equiv B_H^{c_{t_\ell}}/\ell$. The following three facts:

(1) $B_H^{r_i^a[1]}/i$ is a subgraph of G_i^a (by Lemma 18),
(2) $v_k^c \in B_H^{r_i^a[1]}/i$ (by definition),
(3) (v_k^c, v_ℓ^d) is an edge in B_i^a/i (by definition),

imply that v_k^c is the last node of $B_H^{r_i^a[1]}/\ell$ (otherwise the edge emanating from v_k^c in $B_H^{r_i^a[1]}/\ell$ excludes (v_k^c, v_ℓ^d) from B_i^a/i). Since $B_H^{r_i^a[1]}/\ell \equiv B_H^{c_{t_\ell}}/\ell$ it holds that v_k^c is last in $B_H^{c_{t_\ell}}/\ell$. By Definition 15, L_ℓ^d is linearized at c_{t_ℓ} (or just before), that is after $r_i^a[1]$. The claim and the lemma follow. □

In the following theorem, we prove that the (1, 1) implementation is correct.

Theorem 22. *Let L_i^a be a logical read action, and let L_j^b be the logical write action which wrote the value returned by L_i^a. L_j^b is the most recent write action linearized before L_i^a.*

Proof. Denote the initial configuration of L_i^a by c_{s_i}. We prove this theorem by induction on i, the *id* of logical readers.

Base: $i = w + 1$.

Let v_j^{b-r}, $r \geqslant 0$, be node *can_tail* computed in L_i^a. Consider the following cases:

Case 1: L_i^a is enclosed-free.

In this case v_j^{b-r} is the last node of B_i^a/i. By Lemma 18, $B_H^{c_{s_i}}/i$ is a subgraph of G_i^a. We show that either L_j^{b-r} is linearized after c_{s_i} or L_j^{b-r} is the most recent write action linearized before c_{s_i}. If $v_j^{b-r} \notin B_H^{c_{s_i}}/i$ then by Lemma 21, L_j^{b-r} is linearized after c_{s_i}. If $v_j^{b-r} \in B_H^{c_{s_i}}/i$ then since $B_H^{c_{s_i}}/i$ is a subgraph of G_i^a, and since v_j^{b-r} is the last node of B_i^a/i, v_j^{b-r} is last in $B_H^{c_{s_i}}/i$. Hence, L_j^{b-r} is the most recent write action linearized before c_{s_i}. Since $L_j^{b-r} \Rightarrow L_j^b$ (if $r > 0$), the proof follows.

Case 2: L_i^a is not enclosed-free.

In this case, v_j^{b-r} is the node with the maximal *id* detected as enclosed in L_i^a. By Theorem 19, L_j^{b-r} is linearized within its execution interval. Since L_j^{b-r} is enclosed within L_i^a, it is linearized after the beginning of the execution interval of L_i^a. Since L_j^{b-r} is node *can_tail* of L_i^a, Lemma 15 implies that L_j^{b-r} is linearized before the joining configuration of L_i^a. Hence L_j^{b-r} is linearized within the execution interval of L_i^a. If $r = 0$ we are done. In particular, if L_i^a loops then *Ret_node* is *can_tail*, $r = 0$ and we are done. Assume L_i^a connects

and $r > 0$. In this case L_j^b is linearized after the beginning of the execution interval of L_i^a (since $L_j^{b-r} \Rightarrow L_j^b$) and before the joining configuration of L_i^a (since v_j^b is the tail node of v_i^a). In this case L_j^b is linearized within the execution interval of L_i^a and Definition 16 implies that L_i^a is linearized by L_j^b. That is, L_j^b is the most recent write action linearized before L_i^a.

Step: Assume correctness for m, $w < m < i$. Now, we prove for i:

Recall that L_j^b be the logical write action which wrote the value returned by L_i^a. If v_j^b is node *Ret_node* of L_i^a then the proof is identical to the proof of the induction base. Otherwise let v_k^c, $w < k < i$, be node *Ret_node* of L_i^a, and let v_k^{c-r}, $r \geqslant 0$, be node *can_tail* computed during L_i^a. In this case L_k^c is a logical read action which also returns the value written by L_j^b. Consider the following cases:

Case 1: L_i^a is enclosed-free.

By the induction assumption, L_j^b is the most recent write action linearized before L_k^c. If L_k^c is linearized within the execution interval of L_i^a then L_i^a is linearized by L_k^c and we are done. Assume that L_k^c is linearized before c_{s_i}. According to Definition 16, L_i^a is linearized at c_{s_i}. By the next claim there exist no write actions linearized after L_k^c and before c_{s_i} and therefore the proof of this case follows.

Claim 6. *Under the conditions of Case* 1, *there exist no write actions linearized after L_k^c and before c_{s_i}.*

Proof of claim. Assume by way of contradiction that there exist some logical write actions linearized after L_k^c and before c_{s_i}. In this case, there exists at least one such lasting write action. Let v_ℓ^d, $\ell \leqslant w$, be the last node of $B_H^{c_{s_i}}/(w+1)$. By this definition, $L_j^b \Rightarrow L_k^c \Rightarrow L_\ell^d$. By Lemma 18, $B_H^{c_{s_i}}/i$ is a subgraph of G_i^a. Since $v_\ell^d \in B_H^{c_{s_i}}/(w+1)$ we get that $v_\ell^d \in G_i^a$. In both of the following cases we reach the required contradiction:

Case 1.1: $v_\ell^d \in B_i^a/i$.

Since v_k^{c-r} is last in B_i^a/i, there is a path from v_ℓ^d to v_k^{c-r} in G_i^a. Therefore, by Theorem 20, $L_\ell^d \Rightarrow L_k^{c-r}$. Since $r \geqslant 0$ we get that $L_\ell^d \Rightarrow L_k^c$, a contradiction.

Case 1.2: $v_\ell^d \notin B_i^a/i$.

In this case G_i^a has some edge (v_o^f, v_m^e), $o < m < \ell \leqslant w$, that excludes the suffix of $B_H^{c_{s_i}}/i$ (and v_ℓ^d) from B_i^a/i. Let c_{t_m} be the joining configuration of v_m^e. First, we show that $v_\ell^d \in H^{c_{t_m}}$: Since (v_o^f, v_m^e) is an edge of G_i^a, Lemma 14 implies that there exists an edge (v_o^{f+r}, v_m^e) in $H^{c_{t_m}}$. Assume that c_{t_m} is before c_{s_i}. Since $v_o^f \in B_H^{c_{s_i}}$, Lemma 17 implies that $r = 0$ and therefore, (v_o^f, v_m^e) excludes v_ℓ^d from $B_H^{c_{s_i}}$, a contradiction. Thus c_{t_m} is after c_{s_i} and $v_\ell^d \in H^{c_{t_m}}$.

By Lemma 14, there exists some $r \geqslant 0$, such that (v_o^{f+r}, v_m^e) is an edge in $H^{c_{t_m}}$. If $r = 0$, (v_o^f, v_m^e) excludes v_ℓ^d from $B_H^{c_{t_m}}$. If $r > 0$, $v_o^{f+r} \in H^{c_{t_m}}$, and by Lemma 17.(2), $v_o^f \notin B_H^{c_{t_m}}$. Since v_o^f is on the path from the root to v_ℓ^d, $v_\ell^d \notin B_H^{c_{t_m}}$. Since $v_\ell^d \in H^{c_{t_m}}$, L_m^e is either linearized at c_{t_m}, or by another node whose joining configuration is after c_{t_ℓ}, the joining configuration of v_ℓ^d, hence, $L_\ell^d \Rightarrow L_m^e$. Since $v_m^e \in B_i^a/i$ and v_k^{c-r} is last

in B_i^a/i, Theorem 20 implies that $L_m^e \Rightarrow L_k^{c-r}$. Therefore, we get $L_j^b \Rightarrow L_\ell^d \Rightarrow L_m^e \Rightarrow L_k^{c-r} \Rightarrow L_k^c$ and in particular $L_j^b \Rightarrow L_m^e \Rightarrow L_k^c$, a contradiction.

Case 2: L_i^a is not enclosed-free.

In this case v_k^{c-r} is the node with the maximal *id* detected as enclosed within L_i^a. Therefore, L_k^{c-r} is linearized after c_{s_i}, the initial configuration of L_i^a. Since $r \geqslant 0$ we get that L_k^c is linearized after c_{s_i}. By the induction assumption, L_j^b is the most recent write action linearized before L_k^c. Since L_i^a is linearized by L_k^c, L_j^b is the most recent write action linearized before L_i^a. To show that L_k^c is linearized before c_{t_i}, note: If L_i^a loops then *Ret_node* is node *can_tail* of L_i^a, $r = 0$ and the proof follows. If L_i^a connects and the tail node of L_i^a is L_k^{c+r}, $r \geqslant 0$ and if $r > 0$ then $L_k^c \Rightarrow L_k^{c+r}$. By Definition 16, $L_k^{c+r} \Rightarrow L_i^a$ and the proof follows. □

6. Concluding remarks

The second and third implementations presented in this paper are two novel label-based implementations of a (w,r)-atomic register with logarithmic space complexity. The second implementation is of type $(1,n)$, its space complexity is $\Theta(\log w)$, its time complexity is $\Theta(w)$ and communication is one sided. The third implementation is of type $(1,1)$, its space complexity is $\Theta(\log n)$, its time complexity is $\Theta(n)$, and communication is two sided. The logarithmic space complexity of both implementations is asymptotically optimal for label-based implementations. We conjecture that the space complexity of any (w,r) register is logarithmic, even for general (non-label-based) implementations. Our proofs are very technical and complex. We view the problem of presenting a formal verification system for register implementation as an important open problem.

Acknowledgments

We thank Paul Vitányi for his tireless efforts to convince us that implementations with sublinear space complexity are worth looking at. We also thank Yael Gafni, and Arie Rudich whose comments on an earlier version have helped us in this presentation. Last but surely not least is John Tromp whose continuous help during this work was invaluable in terms of both correctness and style.

Appendix A. The hand-shake mechanism

In this appendix we describe the hand-shake mechanism in detail and prove that it satisfies the requirements of Lemma 13. The code of the protocol in which the mechanism is integrated appears in Fig. 13. For each pair of processors, P_j and P_i, $i > j$, the protocol requires that P_i should be able to detect enclosed actions of P_j. To do that, $REG_{j,i}$ is augmented with two bits called $REG_{j,i}.s_1$ and $REG_{j,i}.s_2$, and $REG_{i,j}$ is augmented with one bit called $REG_{i,j}.t$. Processor P_i keeps

```
begin
    for j := 1 to i − 1 do (down block)
        ℓT[j] := ¬ read(REG_{j,i}.S_1)                                   r_i^a[j].s
        write ℓT[j] to REG_{i,j}.t                                       w_i^a[j].t
    endfor
    for j := i + 1 to n do (up block)
        ℓt[j] := read(REG_{j,i}.t)                                       r_i^a[j].t
        if (ℓt[j] ≠ ℓs_1[j]) then
            ℓs_1[j] := ℓt[j];   ℓs_2[j] := ¬ℓt[j]
            write ℓs_1[j] and ℓs_2[j] to REG_{i,j}                       w_i^a[j].s
        else (ℓt[j] = ℓs_1[j])
            ℓs_2[j] := ℓs_1[j]
        endif
    endfor
    collect(G_i, AD, ℓS_1, ℓS_2)                                         r_i^a[1] .. r_i^a[n]
    AD := AD ∪ {current.address, previous.address, old.address, ancient.address}
    enclosed-free := true
    for j := 1 to i − 1 do
        if ℓS_1[j] = ℓS_2[j] = ℓT[j] then
            enclosed-free := false
            max_enclosed := node with maximal id enclosed in L_i^a
        endif
    endfor
    if enclosed-free then
        can_tail := last node in B_i/i
    else
        can_tail := max_enclosed
    endif
        :
        :
        :
        :
        :
    for j := 1 to i
        REG_{i,j} := write (new, current, previous, old, ancient)        w_i^a[1] ... w_i^a[i]
    endfor
    for j := i + 1 to n
        REG_{i,j} := write (ℓs_2[j], new, current, previous, old, ancient)   w_i^a[i + 1] ... w_i^a[n]
    endfor
    if you are a reader (i > w) then return(Ret_node)
end
```

Fig. 13. The protocol for P_i with the hand-shake mechanism.

a local image for each of these bits where the local image of $REG_{j,i}.s_1$ ($REG_{j,i}.s_2$, respectively) is denoted by $\ell s_1[j]$ ($\ell s_2[j]$, respectively) while the local image of $REG_{i,j}.t$ is denoted by $\ell t[j]$. All s bits and their local images are initialized to 0, while all t bits and their local images are initialized to 1.

The detection mechanism works as follows: At the beginning of L_i^a, P_i initiates detection of actions of P_j, $i > j$, by reading bit $REG_{j,i}.s_1$ and writing its inverted value in $REG_{i,j}.t$. Whenever P_i

wishes to check whether there exists an action of P_j, enclosed within L_i^a, P_i reads $REG_{j,i}.s_1$ and $REG_{j,i}.s_2$. In case both bits are equal to bit $REG_{i,j}.t$, the *current* node of P_j is detected as enclosed. Now, we describe the role of P_j in the detection mechanism: Assume that P_j executes L_j^b. P_j always tries to modify the value of $REG_{j,i}.s_1$ (and its local image) so that it is equal to the most recently read value of $REG_{i,j}.t$. At the beginning of L_j^b, P_j reads $REG_{i,j}.t$ and compares its value with (its local image of) $REG_{j,i}.s_1$. If the bits are not equal, P_j concludes that this is the first time $REG_{i,j}.t$ is read since it was last written, thus it is possible that L_i^a has started after L_j^b. In this case P_j assigns $REG_{j,i}.t$ to $REG_{i,j}.s_1$ and the inverted value of $REG_{j,i}.t$ to $REG_{i,j}.s_2$ and prevents P_i from detecting L_j^b as enclosed during L_i^a. On the other hand, if $REG_{i,j}.t$ and (the local image of) $REG_{j,i}.s_1$ are equal, P_j concludes that L_i^a has started before L_j^b and the current value of $REG_{j,i}.s_1$ was written during L_j^{b-1}. In this case, P_j can conclude that L_i^a has started before the end of L_j^{b-1}, and hence before the beginning of L_j^b. In this situation, P_j assigns the value of $REG_{i,j}.s_1$ to $\ell s_2[j]$. This updated value of $\ell s_2[j]$ is written in $REG_{i,j}.s_2$ together with v_j^b in action $w_j^b[i].t$.

Note. If the detection part of L_i^a occurs before this action, L_i^a does not detect any action of P_j as enclosed. If, however, the detection part occurs after this action, the *current* node of P_j is detected as enclosed.

The modified code appears in Fig. 13. For convenience we denote P_i's local images of bits which enable P_i to detect enclosed actions (of processors with lower *id*s) with capital letters while local images of bits which help other processors (with higher *id*s) to detect actions of P_i as enclosed are denoted with lower case letters. To implement the mechanism, the code is modified in four different spots. The first change is to add two blocks of code called *down* and *up* before G_i^a is collected. In block *down*, P_i initiates detection of actions (of processors with lower *id*s) enclosed within L_i^a. For each $j < i$, P_i reads bit $REG_{j,i}.s_1$ (action $r_i^a[j].s$) and writes its inverted value in $REG_{i,j}.t$ and in $\ell t[j]$ (action $w_i^a[j].t$). In block *up*, P_i enables detection of v_i^a as enclosed within the actions of processors with higher *id*s. In this block, P_i computes vectors ℓs_1, ℓs_2, and ℓt; for each $j > i$, P_i reads $REG_{j,i}.t$ (action $r_i^a[j].t$), stores it in $\ell t[j]$ and compares it with $\ell s_1[j]$. If the bits are not equal, P_i assigns $REG_{j,i}.t$ and its inverted value to $REG_{i,j}.s_1$ and to $REG_{i,j}.s_2$, respectively (action $w_i^a[j].s$). If the bits are equal, P_i assigns the value of $\ell s_1[j]$ to $\ell s_2[j]$ (and action $w_i^a.s$ is skipped).

The second change is the augmentation of procedure *collect* so that it returns vectors ℓS_1 and ℓS_2 where $\ell S_1[j]$ ($\ell S_2[j]$) contains bit $REG_{j,i}.s_1$ ($REG_{j,i}.s_2$), read in action $r_i^a[j]$. The third change is in computing predicate *enclosed-free* using vectors ℓS_1, ℓS_2 and ℓT. Node v_j^b is detected as enclosed by L_i^a if bits $REG_{j,i}.s_1$ and $REG_{j,i}.s_2$ read in $r_i^a[j]$ are both equal to bit $REG_{i,j}.t$ stored in $\ell T[j]$ during block *down* (and written in $w_i^a[j].t$). If no enclosed action is detected then *enclosed-free* is true. The fourth change is to write bit $\ell s_2[j]$, for every $j > i$, computed in block *up*, in $REG_{i,j}.s_2$ in action $w_i^a[j]$. The additional physical action required by the hand-shake mechanism are all executed in block *up* and *down* and are denoted by $r_i^a[1].s$; $w_i^a[1].t \cdots r_i^a[i-1].s$; $w_i^a[i-1].t$ and $r_i^a[i+1].t$; $w_i^a[i+1].s \cdots r_i^a[n].t$; $w_i^a[n].s$.

In the next two lemmas we prove that the hand-shake mechanism satisfies the requirements. In Lemma 23, we prove that every label detected as enclosed satisfies Definition 14. In Lemma 24 we prove that the implemented mechanism satisfies the requirements stated in Lemma 13:

Lemma 23. *Let v_j^b be the current node read from $REG_{j,i}$ in action $r_i^a[j]$, where $i > j$. If $REG_{j,i}.s_1$ and $REG_{j,i}.s_2$(the bits read in $r_i^a[j]$) are equal to $REG_{i,j}.t$(the bit written in $w_i^a[j].t$) then v_j^b is enclosed within L_i^a.*

Proof. Bits s_1 and s_2 are always written and read together. Since $REG_{j,i}.s_1 = REG_{j,i}s_2$, we conclude that after P_j read bit $REG_{i,j}.t$, during the *up* block of L_j^b, P_j found that $\ell s_1[i] = \ell t[i]$. Since in this case P_j does not change bit $REG_{j,i}.s_1$, we conclude that the value of $REG_{j,i}.s_1$ was not changed after action $w_j^{b-1}[i].s$. By the code, the value of bit $REG_{i,j}.t$, written in action $w_i^a[j].t$ is the complement of the value read from bit $REG_{j,i}.s_1$ in action $r_i^a[j].s$, we conclude that $r_i^a[j].s \to w_j^{b-1}[i].s$. P_j works sequentially, therefore we get $w_j^{b-1}[i].s \to r_j^b[1].s$. Since v_j^b is read in $r_i^a[j]$ we get $w_j^b[i] \to r_i^a[j]$. Hence $r_i^a[1].s \to r_j^b[1].s \to w_j^b[i] \to r_i^a[j]$ and by Definition 14, v_j^b is enclosed within L_i^a. □

Lemma 24. *Let v_j^b be the last node of P_j whose joining configuration is before $r_i^a[1]$. If $v_j^{b+3+r}, r \geqslant 0$, is the current node in $REG_{j,i}$ at $r_i^a[j]$, then P_i detects L_j^{b+3+r} as an enclosed action.*

Proof. Since v_j^b is the last node of P_j that joined the history graph before $r_i^a[1]$, it holds that $r_i^a[1] \to w_j^{b+1}[n]$. According to the protocol, $w_i^a[j].t \to r_i^a[1]$. Therefore, we get that $w_i^a[j].t \to w_j^{b+1}[n] \to r_j^{b+2}[i].t$. Hence, $w_i^a[j].t \to r_j^{b+2}[i].t \to r_j^{b+3}[i].t$. According to the hand-shake protocol, P_j copies the t bit read in $r_j^{b+2}[i].t$ to $REG_{j,i}.s_1$ (in $w_j^{b+2}[i].s$) and $REG_{j,i}.s_2$ (in $w_j^{b+3}[i]$). Therefore, when executing $r_i^a[j]$, P_i finds $ls_1[j] = ls_2[j] = lT[j]$, hence P_i detects L_j^{b+3} as enclosed. □

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory, J. ACM 40 (4) (1993) 873–890.

[2] U. Abraham, On interprocess communication and the implementation of a multi-writer atomic registers, Theor. Comput. Sci. 149 (1995) 257–298.

[3] R. Cori, E. Sopena, Some combinatorial aspects of time stamp systems, Eur. J. Combin. 14 (1993) 95–102.

[4] D. Dolev, N. Shavit, Bounded concurrent time-stamping, SIAM J. Comput. 26 (2) (1997) 418–455.

[5] M.P. Herlihy, Impossibility and universality results for wait-free synchronization, in: Proceedings of the 7th ACM Symposium on Principles of Distributed Computing, 1988, pp. 276–290.

[6] M. Herlihy, J. Wing, Axioms for concurrent objects, in: 14th ACM Symposium on Principles of Programming Languages, 1987, pp. 13–26.

[7] A. Israeli, M. Li, Bounded time-stamps, Distrib. Comput. 6 (4) (1993) 205–209.

[9] A. Israeli, J. Tromp, P.M.B. Vitányi, personal communication.

[10] L. Lamport, On interprocess communication. Part I: basic formalism, Distrib. Comput. 1 (2) (1986) 77–85.

[11] L. Lamport, On interprocess communication. Part II: algorithms, Distrib. Comput. 1 (2) (1986) 86–101.

[12] N. Lynch, M. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, 1988, pp. 137–151.

[13] M. Li, J. Tromp, P.M.B. Vitányi, How to share concurrent wait-free variables, J. ACM 43 (1992) 107–112.

[14] M. Li, P.M.B. Vitányi, Optimality of wait-free atomic multiwriter variables, Inform. Process. Lett. 43 (1992) 107–112.

[15] J. Misra, Axioms for memory access in asynchronous hardware systems, ACM Trans. Progr. Lang. Syst. 8 (1) (1986) 142–153.

[16] G.L. Peterson, Concurrent reading while writing, ACM Trans. Progr. Lang. Syst. 5 (1) (1983) 46–55.

[17] G.L. Peterson, J.E. Burns, Concurrent reading while writing II: the multiwriter case, in: 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 383–392.
[18] R.W. Schaffer, On the correctness of atomic multi-writer registers, Technical Report MIT/LCS/TM-364, Laboratory for Computer Science, MIT.
[19] J. Tromp, On update-last schemes, Parallel Process. Lett. 44 (1) (1993) 25–28.
[20] P. Vitányi, B. Awerbuch, Atomic shared register access by asynchronous hardware, in: Proceedings of the 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 233–243.