Bounded Concurrent Timestamp Systems Using Vector Clocks

SIBSANKAR HALDAR

Bell Laboratories, Murray Hill, New Jersey

AND

PAUL VITÁNYI

Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands

Abstract. Shared registers are basic objects used as communication mediums in asynchronous concurrent computation. A concurrent timestamp system is a higher typed communication object, and has been shown to be a powerful tool to solve many concurrency control problems. It has turned out to be possible to construct such higher typed objects from primitive lower typed ones. The next step is to find efficient constructions. We propose a very efficient wait-free construction of bounded concurrent timestamp systems from 1-writer shared registers. This finalizes, corrects, and extends a preliminary bounded multiwriter construction proposed by the second author in 1986. That work partially initiated the current interest in wait-free concurrent objects, and introduced a notion of discrete vector clocks in distributed algorithms.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)—*asynchronous/synchronous operation*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming*; D.4.4 [**Operating Systems**]: Communications Management—*buffering*

General Terms: Algorithms, Theory, Verification

This research was supported in parts by the Netherlands Organization for Scientific Research (NWO) under Contract Number NF 62-376 (NFI project ALADDIN), EU Fifth Framework project QAIP, IST-1999-11234, the NoE QUIPROCONE IST-1999-29064, the ESF QiT Programme, and the EU Fourth Framework BRA NeuroCOLT II Working Group EP 27150.

The work of S. Haldar was performed while visiting the Department of Computer Science, Utrecht University, the Netherlands with support from the Netherlands Organization for Scientific Research (NWO) under Contract Number NF 62-376 (NFI project ALADDIN), and continued while he was at the Tata Institute of Fundamental Research, Mumbai, India.

P. Vitányi is also affiliated with the University of Amsterdam.

Author's present addresses: S. Haldar, TimesTen Performance Software, 1991 Landings Drive, Mountain View, CA 94043; P. M. B. Vitányi, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, e-mail: paulv@cwi.nl.

Additional Key Words and Phrases: Concurrent reading while writing; label; nonatomic operation execution; operation—read and write, labeling and scan; operation execution; shared variable—safe, regular and atomic; timestamp system, traceability, vector clock, wait-freedom

1. *Introduction*

Consider a system of asynchronous processes that communicate among themselves by executing read and write operations on a set of shared variables (also known as shared *registers*) only. The system has no global clock or other synchronization primitives. Every shared variable is associated with a process (called *owner*) which writes it and the other processes may read it. An execution of a write (read) operation on a shared variable will be referred to as a *Write* (*Read*) on that variable. A Write on a shared variable puts a value from a predetermined finite domain into the variable, and a Read reports a value from the domain. A process that writes (reads) a variable is called a *writer* (*reader*) of the variable.

1.1. WAIT-FREE SHARED VARIABLE. We want to construct shared variables in which the following two properties hold. (1) Operation executions are not necessarily atomic, that is, they are not indivisible but rather consist of atomic sub-operations, and (2) every operation finishes its execution within a bounded number of its own steps, irrespective of the presence of other operation executions and their relative speeds. That is, operation executions are *wait-free*. These two properties give rise to a classification of shared variables, depending on their output characteristics. Lamport [1986] distinguishes three categories for 1-writer shared variables, using a precedence relation on operation executions defined as follows: for operation executions A and B, A *precedes* B, denoted $A \rightarrow B$, if A finishes before B starts; A and B *overlap* if neither A precedes B nor B precedes A. In 1-writer variables, all the Writes are totally ordered by "$\rightarrow$". The three categories of 1-writer shared variables defined by Lamport are the following:

(1) A *safe* variable is one in which a Read not overlapping any Write returns the most recently written value. A Read that overlaps a Write may return any value from the domain of the variable.
(2) A *regular* variable is a safe variable in which a Read that overlaps one or more Writes returns either the value of the most recent Write preceding the Read or of one of the overlapping Writes.
(3) An *atomic* variable is a regular variable in which the Reads and Writes behave as if they occur in some total order which is an extension of the precedence relation.

A shared variable is *Boolean*[1] or *multivalued* depending upon whether it can hold only two or more than two values.

1.2. MULTIWRITER SHARED VARIABLE. A multiwriter shared variable is one that can be written and read (concurrently) by many processes. Lamport [1986] constructed a shared variable that could be written by one process and read by one other process, but he did not consider constructions of shared variables with more than one writer or reader. Vitányi and Awerbuch [1986] were the first to construct an

[1] Boolean variables are referred to as *bits*.

atomic multiwriter shared variable from 1-writer variables. They propose two constructions: one from 1-writer multireader shared variables using bounded control information that turned out to be incorrect [Vitányi and Awerbuch 1987] (just regular and not atomic as claimed), and the other from 1-writer 1-reader variables using unbounded control information. The latter construction is correct. It is made bounded in Li et al. [1996], yielding one of the most optimal implementations that are currently known. (In this article, we correct and extend the first construction to obtain an efficient version of the more general notion of bounded concurrent timestamp system as defined below.) Related work is Abraham [1995], Bloom [1987/1988], Burns and Peterson [1987], Haldar and Vidyasankar [1995a, 1991/1995b, 1996], Israeli and Shaham [1992], Kirousis et al. [1987], Lamport [1986], Li and Vitányi [1992], Li et al. [1987/1996], Newman-Wolfe [1987], Peterson [1983], Peterson and Burns [1987], Schaffer [1988], Singh et al. [1987/1994], and Vidyasankar [1990]. In particular, it is now possible to construct bounded multiwriter atomic variables from 1-writer 1-reader safe bits. See Li et al. [1996], and the last section of this paper, for a brief history of the subject.

1.3. TIMESTAMP SYSTEM. In a multiwriter-shared variable, it is only required that every process keeps track of which process wrote last. There arises the general question whether every process can keep track of the order of the last Writes by all processes. This idea was formalized by Israeli and Li [1993]. They introduced and analyzed the notion of *timestamp system* as an abstraction of such a higher typed communication medium. In a timestamp system, every process owns an *object*, an abstraction of a set of shared variables. One of the requirements of the system is to determine the temporal order in which the objects are written. For this purpose, each object is given a *label* (also referred to as *timestamp*) which indicates the latest (relative) time when it has been written by its owner process. The processes assign labels to their respective objects in such a way that the labels reflect the real-time order in which they are written to. These systems must support two operations, namely *labeling* and *scan*. A labeling operation execution (Labeling, for short) assigns a new label to an object, and a scan operation execution (Scan, for short) enables a process to determine the ordering in which all the objects are written, that is, it returns a set of labeled-objects ordered temporally. We are concerned with those systems where operations can be executed *concurrently*, in an overlapped fashion. Moreover, operation executions must be *wait-free*, that is, each operation execution will take a bounded number of its own steps (the number of accesses to the shared space), irrespective of the presence of other operation executions and their relative speeds.

Wait-free constructions of concurrent timestamp systems (CTSs, for short) have been shown to be a powerful tool for solving concurrency control problems such as *fcfs*-mutual exclusion [Dijkstra 1965; Lamport 1974], multiwriter multireader shared variables [Vitányi and Awerbuch 1986], probabilistic consensus [Abrahamson 1988; Chor et al. 1987], *fcfs l*-exclusion [Fischer et al. 1979] by synthesizing a "wait-free clock" to sequence the actions in a concurrent system.

Here, we are interested in constructing concurrent timestamp systems using 1-writer shared variables. It is not difficult to construct a timestamp system if the shared space is unbounded (there is no limit on the size of some shared variables). The problem gets much harder for bounded (shared space) systems. A *bounded timestamp system* is a timestamp system with a finite set of bounded size labels.

In the rest of the article, unless stated otherwise, by a timestamp system we mean a wait-free bounded concurrent timestamp system.

Israeli and Li [1987/1993] constructed a bit-optimal bounded timestamp system for sequential operation executions. The *concurrent* case of bounded timestamp system is harder and the first generally accepted solution is due to Dolev and Shavit [1989/1997]. Their construction is the same type as Israeli and Li [1987/1993] and uses shared variables of size $O(n)$, where n is the number of processes in the system. Each Labeling requires $O(n)$ steps, and each Scan $O(n^2 \log n)$ steps. In their construction, no Scan writes any shared variables: It is a "pure" reading operation execution. (But, by the theorem of Lamport [1986, page 91], all such constructions become de facto impure if we break them down to the lowest level of system building.) Following Dolev and Shavit [1997], several researchers have come up with other constructions. Israeli and Pinhasov [1992] use shared variables of size $O(n^2)$; Labeling and Scan require $O(n)$ steps. Gawlik et al. [1992] use shared variables of size $O(n^2)$; Labeling and Scan access $O(n \log n)$ shared variables. Dwork and Waarts [1992/1999] introduce a powerful communication abstraction called "traceable use abstraction" to recycle values of shared variables. They demonstrate the usefulness of the abstraction by constructing a CTS, borrowing the basic ideas and techniques from Vitányi and Awerbuch [1986] for recycling private values. Their construction requires shared variables of size $O(n \log n)$; Labeling and Scan require $O(n)$ steps. Later, they along with Herlihy and Plotkin [Dwork et al. 1992/1999] propose a construction using shared variables of size $O(n)$; Labeling and Scan access $O(n)$ shared variables. Unlike the Israeli–Li and Dolev–Shavit constructions, Scans in other proposed constructions are not pure; they write a lot of shared space.

1.4. OUR RESULT AND RELATED WORK. Among the constructions mentioned above, the one of Dwork and Waarts [1992/1999] is relatively simple and efficient as well.[2] They introduce "traceable-use abstraction" to bound the size of labels. As in Vitányi and Awerbuch [1986], each label is a vector of n private values, one for each of n processes. Using a strategy similar to, and extending Vitányi and Awerbuch [1986], the abstraction helps each process to keep track of its private values that are in use in the system. At any point in time, a process can use only a bounded number of private values of another process. Exploiting that feature, the abstraction helps in bounding the set of private values needed. The labels are read by executing a *traceable-read* function, and written by executing a *traceable-write* procedure. When the traceable-read function is executed to read a label, the executing process explicitly informs all other processes which of their private values it is going to use. A process can find which of its private values are in use by other processes even if the values propagate through these processes in tandem one after another. To determine which of its private values are currently not in use, a process executes a *garbage collection* routine. This routine helps processes to safely recycle their respective private values that are not in use. These three routines are at the heart of implementing the traceable-use abstraction. Dwork and Waarts [1999] have shown how these routines are used in constructing a bounded concurrent timestamp system. The most intricate among these routines is the garbage collection, whose time complexity is $O(n^2)$ that could be, though nonstandard, uniformly amortized

[2] We find it is the easiest one to understand; also see comments by Yakovlev [1993].

over $O(n^2)$ labeling operation executions. To achieve this, each process needs to maintain a private, separate, pool of $22n^2$ private values. The costliest part of their construction is the use of multireader "order" variables. The construction uses, for each process, $\Theta(n)$ sets of $22n$-many 1-writer n-reader atomic variables of size $\Theta(n \log n)$ bits each. Let us roughly estimate their space complexity at the fundamental level, that is, at the level of 1-writer 1-reader safe bits. (To implement a 1-writer n-reader atomic variable of size m bits, the constructions in Lamport [1986] and Vidyasankar [1990] together require $3mn$ 1-writer 1-reader safe bits, $2n$ 1-writer 1-reader atomic bits and one 1-writer n-reader atomic bit. Each 1-writer 1-reader atomic bit can be implemented from $O(1)$ 1-writer 1-reader safe bits [Haldar and Vidyasankar 1995a; Lamport 1986; Tromp 1989; Vidyasankar 1996]. A 1-writer n-reader atomic bit can be implemented from $O(n^2)$ safe bits [Haldar and Vidyasankar 1995a]. Thus, we require a total of $3mn + O(n^2)$ 1-writer 1-reader safe bits to implement a 1-writer n-reader atomic variables of size m bits.) Thus, there is a need of at least $\Omega(n^4 \log n)$ bits at the fundamental level just for the order variables in each process. Consequently, we need at least $\Omega(n^5 \log n)$ 1-writer 1-reader safe bits for all order variables of all processes. In addition, there are other shared variables for the processes.

The bounded multiwriter shared variable construction of Vitányi and Awerbuch [1986], while falling short of the claimed atomicity [Vitányi and Awerbuch 1987], has brought into prominence many techniques that were used later in wait-free computing. An example is the idea of a label as a vector of n individual clocks.[3] (In Vitányi and Awerbuch [1986], vector entries are called "tickets.") Even better, it turns out that the corrected version presented here suffices to implement the higher communication object type of bounded CTS. The current article is the final version of the pioneering preliminary article [Vitányi and Awerbuch 1986] and its correction [Haldar 1993]. Dwork and Waarts [1992/1999] without giving proper credit, used the idea of (bounded) vector clocks and other techniques introduced in Vitányi and Awerbuch [1986], and hence their solution bears a close resemblance to the construction proposed here (and, in fact, to other constructions [Peterson and Burns 1987; Schaffer 1988] based on Vitányi and Awerbuch [1986]). On the other hand, our construction uses some ideas from their traceable-use abstraction. We observe that, in CTSs, the propagation of private values is restricted to only one level of indirection, and not to arbitrary levels. Consequently, the propagation of private values can be tracked down by their respective owner processes with relative ease. And, the one-level indirect propagation of private values by other processes need not be informed to the original owner of these private values. Thus, one doesn't need the complete power of the traceable-use abstraction for constructing a CTS. In our construction, we use less powerful traceable-read and traceable-write. But, we prefer to use the same function/procedure names of Dwork and Waarts [1992/1999] just to keep conformity with the literature. We do not require a garbage collection routine, thereby simplifying the proposed CTS construction and its correctness proof considerably. When a process executes the traceable-read function, it does not explicitly inform the other processes which of their private values it is going to use. On the other hand, the executers of the traceable-write procedure correctly find

[3] The concept of vector clock is used in many areas of distributed computing, all in related contexts, to keep track of execution evolution in distributed systems. (Cf. the articles by Mattern [1989, 1992].)

which private values of which processes are in use in the system. Another important point is that, in our construction, a Scan writes a limited amount of information, only $O(n)$ 1-writer 1-reader bits. Also, each local pool of private values contains fewer than $2n^2$ values. We use a total of n^2 $O(n \log n)$ bit size 1-reader 1-writer regular order variables, requiring a total of $O(n^3 \log n)$ safe 1-reader 1-writer bits at the fundamental level. Both the scan and labeling operation executions require $O(n)$ steps in terms of the shared variables used. But in our construction, a Scan reads at most $(n-1)$ 1-writer 1-reader regular order variables, whereas in their construction it is $(2n-2)$ 1-writer n-reader atomic ones. Thus, at the fundamental level, they scan order-of-magnitude more bits than we do.

Our construction is not optimal in terms of the usage of shared space (Cf. Table I in Section 5). It is perhaps possible to use a bounded set of global values and to recycle them instead of using private values. Recycling of global values could lead to an optimal construction.

The remainder of this article is organized as follows: Section 2 discusses the system model and presents the problem statement precisely. A new construction of concurrent timestamp systems is presented in Section 3, and its correctness proof in Section 4. Section 5 concludes the article.

2. *Model, Problem Definition, and Some Notations*

A concurrent bounded timestamp system (CTS, in short) is an abstract communication system for n completely asynchronous processes $P_1, \ldots, P_n$. It consists of n objects $\mathcal{O}[1..n]$, each of finite space representation, and supports two operations, namely *labeling* and *scan*(*ing*). A labeling operation execution (Labeling, for short) of process P_p assigns a new label to object $\mathcal{O}[p]$. It may use all existing labels of $\mathcal{O}[1..n]$, but it is not allowed to change the labels of components other than $\mathcal{O}[p]$. A scan operation execution (Scan, for short) enables a process to determine the ordering in which all the objects are written, that is, it returns a set of labeled-objects ordered temporally.[4] It returns a pair $(\underline{l}, \prec)$, where $\underline{l}$ is a set of current labels, one for each object-component, and $\prec$ is a total order on $\underline{l}$. Operation executions of each process are sequential. However, operation executions of different processes need not be sequential, that is, they might overlap.

Let us denote the kth operation execution (Labeling or Scan) of a process P_p by $O_p^{[k]}$, $k \geq 1$. If it is a Scan (Labeling), we denote it explicitly by $S_p^{[k]}$ ($L_p^{[k]}$). The label written by a labeling operation execution $L_p^{[k]}$ is denoted by $l_p^{[k]}$.

For operation executions A and B on a shared variable, $A \dashrightarrow B$ means that the execution of A starts before that of B finishes. That is, if $A \dashrightarrow B$, then either $A \rightarrow B$ or A overlaps B; in other words, $B \not\rightarrow A$. We also assume that if $B \not\rightarrow A$, then $A \dashrightarrow B$. That is, we assume the global time model [Lamport 1986].

A concurrent timestamp system must ensure the following properties [Dolev and Shavit 1997; Gawlick et al. 1992]:

P1. *Ordering.* There exists an irreflexive total order $\Rightarrow$ on the set of all labeling operation executions, such that the following two conditions hold.
—*Precedence*. For every pair of Labelings $L_p^{[k]}$ and $L_q^{[k']}$, if $L_p^{[k]} \rightarrow L_q^{[k']}$ then $L_p^{[k]} \Rightarrow L_q^{[k']}$.

[4] We ignore, in this article, the data values of the objects.

—*Consistency.* For every Scan $S_i^{[j]}$ returning $(\bar{l}, \prec)$, for every two labels $l_p^{[k]}$ and $l_q^{[k']}$ in $\bar{l}$, $l_p^{[k]} \prec l_q^{[k']}$ iff $L_p^{[k]} \Rightarrow L_q^{[k']}$.

P2. *Regularity.* For every label $l_p^{[k]}$ in $\bar{l}$ returned by a Scan $S_i^{[j]}$, $L_p^{[k]}$ begins before $S_i^{[j]}$ terminates, that is, $L_p^{[k]} \dashrightarrow S_i^{[j]}$, and there is no Labeling $L_p^{[k']}$ such that $L_p^{[k]} \rightarrow L_p^{[k']} \rightarrow S_i^{[j]}$.

P3. *Monotonicity.* Let $S_i^{[j]}$ and $S_{i'}^{[j']}$ be a pair of Scans returning sets $\bar{l}$ and $\bar{l}'$, respectively, which contain labels $l_p^{[k]}$ and $l_p^{[k']}$, respectively. If $S_i^{[j]} \rightarrow S_{i'}^{[j']}$, then $k \leq k'$.

P4. *Extended Regularity.* Let $l_p^{[k]}$ be a label returned by a Scan $S_i^{[j]}$. For each Labeling $L_q^{[k']}$, if $S_i^{[j]} \rightarrow L_q^{[k']}$, then $L_p^{[k]} \Rightarrow L_q^{[k']}$.

The intuitive meaning of the above four properties is as follows: The ordering property says that all the labeling operation executions can be totally ordered, which is an extension of their real-time precedence order "$\rightarrow$". Moreover, if two different Scans return labels l and l', then both Scans will have the same order on the labels. The regularity property says that labels returned by a Scan are not obsolete. The monotonicity property says that for every two Scans ordered by "$\rightarrow$", it is not the case that the preceding Scan returns a new label of a process P_p and the succeeding Scan an old label of P_p. The monotonicity property does not imply that labeling and scan operation executions of all processes are linearizable [Herlihy and Wing 1990]. It does imply the linearizability of the Scans of all processes and labeling operation executions of a single process [Dolev and Shavit 1997]. The extended regularity property says that if a Scan precedes a labeling operation execution L, then all labels returned by the Scan were assigned by labeling operation executions that precede L in $\Rightarrow$.

We are interested in those CTSs in which operation executions are *wait-free*, that is, each operation execution will take a bounded number of its own steps (a step is a read/write of a shared variable), irrespective of the presence of other operation executions and their relative speeds. This article is concerned with implementing wait-free CTSs from basic 1-writer 1-reader shared variables.

3. *The Construction*

For the sake of convenience and better understanding, we first present an intuitive informal description of a construction that uses unbounded shared space [Vitányi and Awerbuch 1986] (the same idea is used in Dwork and Waarts [1992/1999]). Each process maintains a separate local pool of private values that are natural numbers with the standard order relations on them.

A label is a vector of n values ("tickets" in Vitányi and Awerbuch [1986]); its pth component holds a private value of process P_p. The current label of $\mathcal{O}[p]$ is denoted by $l_p[1..n]$ or simply l_p. The current private value of process P_p is $l_p[p]$. Initially, $l_p[p] = 1$ and $l_p[q] = 0$, for all $q \neq p$. To determine a new label for $\mathcal{O}[p]$, process P_p reads all current private values of other processes P_q, namely, $l_q[q]$, and increments its own private value $l_p[p]$ by one to obtain the new private value. The new label vector contains these n values, and it is written atomically in $\mathcal{O}[p]$. Since the same private value is not used twice in labeling operation executions, no two labels ever produced in the system are the same. The ordering of two label vectors is done by using the standard lexicographic (dictionary) order $\prec$: for every two

labels, $l_p \neq l_q$, the *least significant index* in which they differ is the lowest k such that $l_p[k] \neq l_q[k]$; then, $l_p \prec l_q$ iff $l_p[k] < l_q[k]$. This lexicographic order $\prec$ is a total order on the set of all possible labels [Fishburn 1985], and this fact is a static common knowledge to the processes. (In fact, $\prec$ is an elementary example of a well-ordered relation.) A Scan simply reads all the current labels and orders them using the lexicographic order. This unbounded construction satisfies all the properties required for a concurrent timestamp system (Cf. [Dwork and Waarts 1992/1999]).

In the unbounded construction discussed above, every time a process P_k executes a new labeling operation, it uses a new private value greater than the previously used ones. In a bounded construction, each process has only a bounded number of private values, and hence, it needs to use the same private value at different times, that is, it needs to recycle its own private values. The following observation (which is a synthesis of the text in Vitányi and Awerbuch [1986, page 236]) by Dwork and Waarts helps doing the recycling in some possible way. We quote them verbatim:

> . . . for a system to be a concurrent timestamp system, every time a new private value chosen by process P_k need not be the one that was never used by P_k beforehand; roughly speaking, instead of increasing its private value, it is enough for P_k to take as its new private value any value v of its private values that does not appear in any labels, with one proviso: P_k must inform the other processes that v is to be considered larger than all its other private values currently in use.

Consequently, we cannot use the standard ordering relations on the natural numbers any more, for the numbers may be recycled repeatedly. One now has to consider these numbers as mere symbols with no standard ordering relations defined on them. We define for every two different private values v and v' of process P_k currently in use in the system, $v \prec_k v'$ iff v is issued before v' by P_k. Thus, in the bounded construction, the ordering relation among the private values changes in time, and hence it cannot be *a priori* common knowledge. Note that at any point in time, the relation $\prec_k$ on the values in use is a total order as the values are produced in sequences, and in fact, it is well ordered. For every two labels, $l_p \neq l_q$, obtained by a Scan, if k is the least significant index such that $l_p[k] \neq l_q[k]$, then we define $l_p \prec l_q$ iff $l_p[k] \prec_k l_q[k]$. Then, $\prec$ is also a well-ordered relation [Fishburn 1985]. Now, we are concerned with two things in a bounded construction. First, to make the relations $\prec_k$ useful, processes P_k cannot recycle a private value if some other processes are using it. Second, for every two private values v and v' of P_k currently in use, if $v \prec_k v'$, then all other processes should (get to) know this ordering before using these values. Note that the meaning of $<$ on the natural numbers is a static common knowledge, but the meaning of $\prec_k$ changes continually. Thus, every time P_k changes the ordering of two different private values, it should inform all the other processes well in advance. Then, for all labels read by a Scan, the labels are ordered lexicographically, based on the orderings $\prec_k$ of all processes P_k. Then, the correctness of the bounded system trivially follows from that of the unbounded system mentioned above (given in Vitányi and Awerbuch [1986] and Dwork and Waarts [1992/1999]).

In the following paragraphs, we present a novel construction, based on Vitányi and Awerbuch [1986] and Haldar [1993] to achieve the aforementioned two objectives. The construction is given in Figure 1.

Declarations

Constants:

n = number of processes;

Type:

label-type: array [1..n] of natural number; {represents vector clock}
boolean: 0..1;

Shared variables and their initial values:

w : array [1..n, 1..n] of boolean *atomic*; {**all initially** 0}
{P_p writes $w[p, 1..n]$ and P_i reads $w[1..n, i]$}

r : array [1..n, 1..n] of boolean *atomic*; {**all initially** 0}
{P_p writes $r[p, 1..n]$ and P_i reads $r[1..n, i]$}

c : array [1..n] of boolean *atomic*; {**initially** 0}
{P_p writes $c[p]$, and the others read}

label : array [1..n, 0..1] of label-type *safe*; {**all initially 0, except** $label[p, 0][p] = 1$ **for all** p}
{P_p writes $label[p, 0..1]$ and the others read}

copylabel : array [1..n, 1..n] of label-type *safe*;
{P_p writes $copylabel[p, 1..n]$ and P_i reads $copylabel[1..n, i]$}

lend: array [1..n, 1..n] of *regular* array [0..1] of label-type; {**all initially** 0}
{P_p writes $lend[p, 1..n]$ and P_i reads $lend[1..n, i]$}

order: array [1..n, 1..n] of *regular* array [1..5n] of natural number;
{**initially** $order[1..n, 1..n][1] = 0$ and $order[1..n, 1..n][2] = 1$}
{P_p writes $order[p, 1..n]$ and P_i reads $order[1..n, i]$}

Private variables for process P_p, $p = 1, 2, \ldots, n$:

cl_p: boolean; {**initially** 0}
$myLend_p$: array [1..n] of array [0..1] of label-type; {**all initially 0**}
$old\text{-}label_p$: label-type; {**all initially 0**, except $old\text{-}label_p[p] = 1$}
$\prec_p$: total order relation; {**initially** $\{\langle 0, 1\rangle\}$}

FIG. 1. Shared variables.

We now introduce some terminology. The description of the construction has five parts: shared variables declaration, TRACEABLE-WRITE procedure, TRACEABLE-READ function, LABELING procedure and SCAN function. The procedures and the functions are written in a Pascal-type language. To avoid too many "begin" s and "end" s, some blocks are shown just by indentation. All the statements in the four routines are numbered only for reference purposes.

A base shared variable x is read (respectively, written) by executing an instruction "read *local-variable* from x" (respectively, "write *local-variable* in x"), where the *local-variable* is local to the function or the procedure. The read-instruction assigns the value of x to the *local-variable*, and the write-instruction writes the value of the *local-variable* in x. The writer (owner) of a shared variable can retain the value of the variable in its local storage and refer to it later on if needed, that is, it need not read the shared variable to determine the current value of the variable. Nevertheless, for the sake of convenience and to avoid using many local variables, we let the writer also read its own shared variable. It also uses some private

```
Procedure TRACEABLE-WRITE(p: 1..n; new-label: label-type); {P_p writes new-label in O[p]}
var
    i, j: 1..n; {loop index}
    lr: boolean;
begin
1.  cl_p := ¬cl_p;
2.  write new-label in label[p, cl_p];
3.  write cl_p in c[p];
4.  for i := 1 to n do
        begin                                   {could be done in parallel}
4.1       read lr from r[i, p];
4.2       if lr ≠ w[p, i] then
4.2.1       write new-label in copylabel[p, i];
4.2.2       for j := 1 to n do myLend_p[j][0..1][i] := ⟨old-label_p[j], new-label[j]⟩;
4.2.3       write lr in w[p, i];                {w[p, i] = r[i, p]}
          endif;
        endfor;
5.  for j := 1 to n do myLend_p[j][1][p] := new-label[j];
6.  for j := 1 to n do write myLend_p[j] in lend[p, j]; {could be done in parallel}
7.  old-label_p := new-label;
end; {of procedure}

Function TRACEABLE-READ(p: 1..n, i: 1..n): label-type; {P_p reads a label from P_i}
var
    lw: boolean;
    lc: boolean;
    savelabel: label-type;
begin
1.  read lw from w[i, p];
2.  write ¬lw in r[p, i];                       {r[p, i] ≠ w[i, p]}
3.  read lc from c[i];
4.  read savelabel from label[i, lc];
5.  read lw from w[i, p];
6.  if (r[p, i] ≠ lw) then return(savelabel)
7.  else                                        {r[p, i] = w[i, p]}
      read and return(copylabel[i, p])
    endif;
end; {of function}
```

FIG. 1. Construction for process P_p. (Cont'd.)

(local, nonshared) variables for each process. We assume that the private variables are persistent.

Let us consider operation executions of a particular process P_p. Process P_p executes the LABELING procedure to obtain and assign a new label to $\mathcal{O}[p]$, and executes the SCAN function to report the temporal ordering of the labels of $\mathcal{O}[1..n]$. In a labeling operation execution, it selects a presently unused private value from its local pool of values (Statements 1–2 in the LABELING procedure), collects the current private values of all other processes (Statements 5–6), and then writes these n values atomically in $\mathcal{O}[p]$ as its new label (Statement 7). The selection of a new private value is done in such a way that there is no trace of this value in the system at present. In a scan operation execution, process P_p first reads the current labels of all the processes (Statement 1 in the SCAN function), and then determines their temporal ordering using the latest ordering information available from some ordering shared variables (Statement 2).

The collection of the current private values of other processes is done by executing the TRACEABLE-READ function, and the writing of the new label is done by

```
Procedure LABELING(p: 1..n);
var
    j, k: 1..n;
    temp: array [1..n] of array [0..1] of label-type;
    lab: array [1..n] of label-type;
    new-label: label-type;
    private-value: natural number;
begin
1.  for j := 1 to n do {could be done in parallel}
        read temp[j] from lend[j, p]; {we do not need temp[j][0][j]}
2.  select a new private-value not in temp[1..n] and the current private value; {use the axiom of choice here}
3.  put the new private-value in ≺_p as the largest element;
4.  for j := 1 to n do {could be done in parallel}
4.1     order the elements of (temp[1..n][0..1][j],
                               temp[k][1][k] and
                               temp[p][0..1][k] for all k,
                               and the new private-value) consistent with ≺_p
4.2     and write them in order[p, j];
5.  for j := 1 to n, j ≠ p, do lab[j] :=TRACEABLE-READ(p, j); {could be done in parallel}
6.  new-label := ⟨lab[1][1], lab[2][2], ..., lab[p][p] := private-value, ..., lab[n][n]⟩;
7.  TRACEABLE-WRITE(p, new-label);
end;

Function SCAN(p: 1..n):(l, ≺);
var
    i, j, k: 1..n;
    lab: array [1..n] of label-type;
begin
1.  for j := 1 to n do lab[j] :=TRACEABLE-READ(p, j); {could be done in parallel}
2.  for i := 1 to n do
2.1     for j := 1 to n do
2.1.1       let k be the least significant index in which lab[i] differs from lab[j];
2.1.2       if order[k, p] (which is a subset of ≺_k) is not read yet, then read it;
2.1.3       determine the order between lab[i] and lab[j] using ≺_k;
end;
```

FIG. 1. Construction for process P_p. (Cont'd.)

executing the TRACEABLE-WRITE procedure.[5] These two routines collectively implement atomic reading and writing of labels from and into objects $\mathcal{O}[p]$. (In rest of the article, an execution of the TRACEABLE-READ function (TRACEABLE-WRITE procedure) will be called a traceable Read (traceable Write).) Note that these two routines are not parts of the interface to the CTS, and the processes cannot directly invoke them. They directly invoke the LABELING and SCAN routines in which they, in turn, invoke traceable Read (Write) to read (write) labels.

A process P_p uses shared variables $w[p, 1..n]$, $r[p, 1..n]$, $c[p]$, *label*$[p, 0..1]$ and *copylabel*$[p, 1..n]$ to read and write new labels from and into object $\mathcal{O}[p]$ atomically. The *label* and *copylabel* variables are used to hold labels of $\mathcal{O}[p]$. w and r are handshake variables used to detect overlapping of traceable Reads and Writes. The variable c is used to atomically declare writings of new labels in $\mathcal{O}[p]$. Process P_p uses the shared variables *order*$[p, 1..n]$ to inform all the processes of the latest ordering relation $\prec_p$. The shared variables *lend*$[p, 1..n]$ are used to inform all the processes which of their private values might be in use in the system.

[5] These two routines resemble the READ and WRITE routines in Haldar and Vidyasankar [1991/1995b, 1996], Vidyasankar [1990], and Vitányi and Awerbuch [1986] pretty closely.

The component $lend[p, j]$ contains all the private values of process P_j that P_p may have lent to other processes. Process P_p also uses static private variables: cl_p, $myLend_p$, $\prec_p$, and $old\text{-}label_p$, cl_p and $myLend_p$ always store the values of $c[p]$ and $lend[p, 1..n]$, respectively, locally. $\prec_p$ contains the latest ordering information of all the private values in use in the system. $old\text{-}label_p$ stores the label of the ongoing or the recently completed Labeling operation execution.

The traceable Writes of process P_p use two n-reader safe *main label variables*, $label[p, 0]$ and $label[p, 1]$, and a 1-reader safe *copy label variable* for each process, $copylabel[p, 1..n]$. The main label variables are used alternately for writing successive new labels. Immediately after writing a new label in a main label variable, the process records that variable index in the 1-writer multireader Boolean atomic variable $c[p]$. (This writing atomically "declares" the current label of component $\mathcal{O}[p]$.) Then, the process checks for each i whether a new traceable Read of process P_i started since the last traceable Write (of P_p). This is done by using a pair of Boolean 1-writer 1-reader (handshaking) atomic variables $r[i, p]$ and $w[p, i]$.[6] Process P_i sets these values different, by assigning the complement of $w[p, i]$ to $r[i, p]$ at the beginning of each traceable Read (Statements 1–2 in TRACEABLE-READ), and process P_p makes sure that they are the same, at the end of each traceable Write (Statements 4.1 and 4.2.3 in TRACEABLE-WRITE). In this way, the processes P_p and P_i can find if there are overlappings of their traceable Writes and Reads. Hence, if the two values are different when the process P_p checks them, a new traceable Read of P_i must have started by then. In that case, P_p writes the new label value in $copylabel[p, i]$ also, and then sets the above values the same, by assigning the $r[i, p]$ value to $w[p, i]$. (This way it is guaranteed that a reading and a writing on *copylabel* variables do not overlap each other, and contain a valid value for the traceable Read [Vitányi and Awerbuch 1986; Haldar and Vidyasankar 1991/1995b; Vidyasankar 1990].) For each such process P_i, P_p takes note of which of the private values of processes P_j could be used by P_i (Statement 4.2.2). Finally, P_p informs all the processes P_j which of their private values could be in use (all that P_p knows of) through 1-writer 1-reader regular variables $lend[p, j]$ (Statement 6).

Each traceable Read of process P_p, from a process P_i, after reading $w[i, p]$ and writing its complement in $r[p, i]$ as mentioned above (Statements 1–2 in TRACEABLE-READ), finds out from $c[i]$ the main label variable that has been written by P_i most recently, and reads from that variable. Then, it reads $w[i, p]$ again and compares with $r[p, i]$. If the two values continue to be different, then the reading of the main label variable does not overlap any writings of the label variable and hence it returns the value just read from the main label variable. Otherwise, there is a possibility that the reading of the label variable overlaps with some writing of the same variable, and hence, it reads $copylabel[i, p]$ and returns that value. Note that, in the latter case, a traceable Write by P_i must have finished (with respect to P_p, i.e., P_i must have done loop iteration p at Statement 4 in TRACEABLE-WRITE) after the traceable Read started, and that Write would have written in $copylabel[i, p]$.

In selecting a new (currently unused) private value, process P_p does not use any of the values stored in $lend[1..n, p]$ (Statements 1–2 in LABELING). After selecting the new private value, say v, P_p informs all processes P_i that v is the most recent private value through 1-writer 1-reader regular variables $order[p, i]$ (Statements 3–4), which are used by the Scans of P_i.

[6] This strategy of detecting overlapping operation execution is pioneered by Peterson [1983].

4. *Correctness Proof*

PROPOSITION 1 [LAMPORT 1986]. *For operation executions B and C on a shared variable, and all operation executions A and D, if $A \to B \dashrightarrow C \to D$, then $A \to D$.*

PROOF. The implication follows by the transitivity of (i) A finishes before B starts, (ii) B starts before C finishes and (iii) C finishes before D starts. □

Definition 1. For operation executions A and B executed on the same atomic variable x, we say $A \Rightarrow_x B$ if A precedes B in the total ordering imposed on the operation executions by the atomic variable. The subscript x is omitted when it is clear from the context.

PROPOSITION 2. *For operation executions B and C on an atomic variable x, and all operation executions A and D, if $A \to B \Rightarrow_x C \to D$, then $A \to D$.*

PROOF. The relation $B \Rightarrow_x C$ implies B precedes or overlaps C (since the total order imposed on the operation executions by the atomic variable is an extension of the precedence relation), that is, $B \dashrightarrow C$. Then the implication follows by Proposition 1. □

The following notations are used in the presentation of the correctness proofs.

N1. The kth operation execution of a process P_p is denoted, as stated in Section 2, by $O_p^{[k]}(\mathcal{O})$, $k \geq 1$; if it is a Scan (alternatively, a Labeling), we denote it explicitly by $S_p^{[k]}(\mathcal{O})$ (alternatively, $L_p^{[k]}(\mathcal{O})$). The "$(\mathcal{O})$" part in the notation is omitted when it is clear from the context. All the operation executions of P_p are totally ordered. That is, for $k > 2$ $O_p^{[k-1]} \to O_p^{[k]}$.

N2. For a shared variable x, the Read (respectively, Write) of x by $O_p^{[k]}$ is denoted by $R_p^{[k]}(x)$ (respectively, $W_p^{[k]}(x)$). If x is referred more than once, then the superscript $[k, j]$ is used for the jth access.

N3. Each operation execution $O_p^{[k]}$ ($L_p^{[k]}$ or $S_p^{[k]}$) of process P_p executes the TRACEABLE-READ function for every other process P_i; the whole function execution is denoted by a traceable Read $TR_{p,i}^{[k]}$.

N4. Each labeling operation execution $L_p^{[k]}$ of process P_p executes the TRACEABLE-WRITE procedure; the whole procedure execution is denoted by a traceable Write $TW_p^{[k]}$.

N5. For the sake of convenience, the variables $r[p, i]$ and $w[p, i]$ are abbreviated to $r_{p,i}$ and $w_{p,i}$, respectively.

Definition 2. For a shared variable x, we define a *reading mapping* π_x for Reads of x as follows: if a Read R returns the value written by a Write W, then $\pi_x(R)$ is W; otherwise, $\pi_x(R)$ is undefined. (Note, for safe x, π_x is a partial mapping.) We omit the subscript x when it is clear from the context.

LEMMA 1

(*a*) *No two consecutive labeling operation executions of a process have the same private value.*

(*b*) *No two consecutive traceable Writes of a process have the same private value.*

PROOF. Part (a) follows from the select statement (Statement 2) in the LABELING procedure. Part (b) follows from Part (a) as each Labeling executes one and the only one traceable Write. □

LEMMA 2. *Each time the value written in $w_{p,i}$ is the complement of the previous value of $w_{p,i}$.*

PROOF. Immediate from Statements 4.1, 4.2 and 4.2.3 in the TRACEABLE-WRITE procedure. □

LEMMA 3. *Any traceable Write $TW_p^{[k]}$ (actually, $L_p^{[k]}$) that writes $w_{p,i}$ sets $w_{p,i} = r_{i,p}$, and if $R_i^{[l,1]}(w_{p,i}) \Rightarrow W_p^{[k]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i})$ for some traceable Read $TR_{i,p}^{[l]}$ (actually, $O_i^{[l]}$) of process P_i, then the equality continues to hold until the execution of $TR_{i,p}^{[l]}$ is complete, in fact until the next traceable Read $TR_{i,p}^{[l+1]}$ writes $r_{i,p}$.*

PROOF. Initially, $w_{p,i} = r_{i,p}$, since both of them are initialized to 0. Among the traceable Writes of the process P_p, some will write $w_{p,i}$, and some will not. Let $TW_p^{[k_j]}$, $j \geq 1$, $k_j \geq 1$, be the jth traceable Write that writes $w_{p,i}$.

Consider $TW_p^{[k_1]}$. By Lemma 2, it writes 1 in $w_{p,i}$. This implies, by Statements 4.1 and 4.2.3 in TRACEABLE-WRITE, that it read 1 from $r_{i,p}$. Since the initial value of $r_{i,p}$ is 0, some traceable Read of P_i must have written 1 in $r_{i,p}$. Let $TR_{i,p}^{[l_1]}$ be the first such traceable Read. Then $W_i^{[l_1]}(r_{i,p}) \Rightarrow R_p^{[k_1]}(r_{i,p})$. Note that $TR_{i,p}^{[l_1]}$ reads 0 from $w_{p,i}$ and hence writes 1 in $r_{i,p}$ (Statements 1–2 in TRACEABLE-READ). Also each subsequent traceable Read $TR_{i,p}^{[l'_1]}$, if any, such that $R_i^{[l'_1,1]}(w_{p,i}) \Rightarrow W_p^{[k_1]}(w_{p,i})$, would read 0 from $w_{p,i}$, and hence will write 1 in $r_{i,p}$. Hence, irrespective of whether $W_i^{[l'_1]}(r_{i,p}) \Rightarrow R_p^{[k_1]}(r_{i,p})$ or $R_p^{[k_1]}(r_{i,p}) \Rightarrow W_i^{[l'_1]}(r_{i,p})$, on $W_p^{[k_1]}(w_{p,i})$, $w_{p,i} = r_{i,p}$, and if $R_i^{[l,1]}(w_{p,i}) \Rightarrow W_p^{[k_1]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i})$ for some traceable Read $TR_{i,p}^{[l]}$, then the equality continues to hold until $TR_{i,p}^{[l]}$ is complete, in fact until the next traceable Read $TR_{i,p}^{[l+1]}$ writes $r_{i,p}$, since $w_{p,i}$ will not be changed by any traceable Write $TW_p^{[k'_1]}$, for $k'_1 > k_1$, that may occur before $TR_{i,p}^{[l]}$ is complete.

Assuming as induction hypothesis that the assertion holds for $TW_p^{[k_j]}$, for some j, we show that the assertion holds for $TW_p^{[k_{j+1}]}$. By the statement of the lemma, $TW_p^{[k_j]}$ sets $w_{p,i} = r_{i,p}$ by writing value, say $b \in \{0, 1\}$ in $w_{p,i}$. Then, by Lemma 2, $TW_p^{[k_{j+1}]}$ writes $\neg b$ in $w_{p,i}$.[7] This implies, by Statements 4.1 and 4.2.3 in TRACEABLE-WRITE, that it read $\neg b$ from $r_{i,p}$. As the value of $r_{i,p}$ is b when $TW_p^{[k_j]}$ reads it, there must be a traceable Read that writes $\neg b$ in $r_{i,p}$ after $TW_p^{[k_j]}$ sets $w_{p,i} = r_{i,p}$. Let $TR_{i,p}^{[l]}$ be the first such traceable Read. Then, $W_i^{[l]}(r_{i,p}) \Rightarrow R_p^{[k_{j+1}]}(r_{i,p})$, and $TR_{i,p}^{[l]}$ writes $\neg b$ in $r_{i,p}$. Each subsequent traceable Read $TR_{i,p}^{[l']}$, if any, such that $R_i^{[l',1]}(w_{p,i}) \Rightarrow W_p^{[k_{j+1}]}(w_{p,i})$, would read b from $w_{p,i}$, and hence will write $\neg b$ in $r_{i,p}$. Hence, irrespective of whether $W_i^{[l']}(r_{i,p}) \Rightarrow R_p^{[k_{j+1}]}(r_{i,p})$ or $R_p^{[k_{j+1}]}(r_{i,p}) \Rightarrow W_i^{[l']}(r_{i,p})$, on $W_p^{[k_{j+1}]}(w_{p,i})$, $w_{p,i} = r_{i,p}$. If $R_i^{[l]}(w_{p,i}) \Rightarrow W_p^{[k_{j+1}]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i})$ for some traceable Read $TR_{i,p}^{[l]}$, then the equality continues to hold until $TR_{i,p}^{[l]}$ is complete, (in fact, until the next traceable Read $TR_{i,p}^{[l+1]}$ writes $r_{i,p}$, since $w_{p,i}$ will not be changed by any traceable Write $TW_p^{[k']}$ that may occur before $TR_{i,p}^{[l]}$ is complete, for $k' > k_{j+1}$.) □

[7] $\neg b$ is defined as $1 - b$.

Lemma 3 implies the following property.

LEMMA 4. *Let $TR_{i,p}^{[l]}$ be a traceable Read. There can be at most one traceable Write, say $TW_p^{[k]}$, such that $R_i^{[l,1]}(w_{p,i}) \Rightarrow W_p^{[k]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i})$. The traceable Read $TR_{i,p}^{[l]}$ on $R_i^{[l,2]}(w_{p,i})$ will find $r_{i,p} = w_{p,i}$ if there is such a traceable Write, and $r_{i,p} \neq w_{p,i}$ otherwise.*

In the following, we use a typical kind of notation for labeling operation executions.

N6. The labeling operation executions of process P_p are sometimes denoted by $L_p^{[k_j]}$, where k is some alphabet and j is a natural number, $j \geq 1, k_j \geq 1$. Thus, for $j > 1$, $L_p^{[k_{j-1}]}$ and $L_p^{[k_j]}$ are two consecutive labeling operation executions of P_p such that $L_p^{[k_{j-1}]} \rightarrow L_p^{[k_j]}$. They need not be two consecutive operation executions, that is, $k_j \geq k_{j-1} + 1$.

In the following two lemmas, we show that traceable Reads return valid label values. We also define their reading mapping function π. Lemmas 5 and 6 deal with the case-traceable Reads return values from *label* and *copylabel* variables, respectively.

LEMMA 5. *Let $TR_{i,p}^{[l]}$ be a traceable Read that finds $r_{i,p} \neq w_{p,i}$ on $R_i^{[l,2]}(w_{p,i})$. Suppose $\pi(R_i^{[l]}(c[p]))$ is $W_p^{[k_j]}(c[p])$ (of the traceable Write $TW_p^{[k_j]}$ of $L_p^{[k_j]}$), and $label[p,x]$ is the main label variable from which $TR_{i,p}^{[l]}$ returns the label value.*

(*a*) *If j' is the least index such that $R_i^{[l,2]}(w_{p,i}) \Rightarrow W_p^{[k_{j'}]}(w_{p,i})$, then j' equals j or $j+1$.*
(*b*) *$\pi(TR_{i,p}^{[l]})$ is $TW_p^{[k_j]}$.*
(*c*) *The traceable Read $TR_{i,p}^{[l]}$ reading $label[p,x]$ does not conflict with any traceable Write writing that label variable.*

PROOF

(*a*) Let j'' be the greatest index such that $j'' < j'$ and $TW_p^{[k_{j''}]}$ writes $w_{p,i}$. Then, by (i) the choice of j', (ii) the assumption that $TR_{i,p}^{[l]}$ finds $r_{i,p} \neq w_{p,i}$ on $R_i^{[l,2]}(w_{p,i})$, and (iii) Lemma 4, it follows that $W_p^{[k_{j''}]}(w_{p,i}) \Rightarrow R_i^{[l,1]}(w_{p,i})$. That is, $W_p^{[k_{j''}]}(w_{p,i}) \Rightarrow R_i^{[l,1]}(w_{p,i}) \rightarrow R_i^{[l,2]}(w_{p,i}) \Rightarrow W_p^{[k_{j'}]}(w_{p,i})$. The traceable Write $TW_p^{[k_{j''}]}$ sets $w_{p,i}$ equal to $r_{i,p}$, $TR_{i,p}^{[l]}$ sets $r_{i,p}$ not equal to $w_{p,i}$, and hence $TW_p^{[k_{j'}]}$ is the first traceable Write, after $TW_p^{[k_{j''}]}$, that finds $r_{i,p} \neq w_{p,i}$.

From $W_i^{[l]}(r_{i,p}) \rightarrow R_i^{[l]}(c[p]) \Rightarrow W_p^{[k_{j+1}]}(c[p]) \rightarrow R_p^{[k_{j+1}]}(r_{i,p})$, we have $W_i^{[l]}(r_{i,p}) \rightarrow R_p^{[k_{j+1}]}(r_{i,p})$. That is, the traceable Write $TW_p^{[k_{j+1}]}$ will find $r_{i,p} \neq w_{p,i}$, the inequality set by $TR_{i,p}^{[l]}$, unless an earlier traceable Write has found the inequality and set $w_{p,i}$ equal to $r_{i,p}$. We claim that such an earlier traceable Write, if one exists, can only be $TW_p^{[k_j]}$. Suppose, on the contrary, that it is $TW_p^{[k_{j'''}]}$, for $j''' < j$. Then, by the choice of j'' and Lemma 4, we have $W_p^{[k_{j''}]}(w_{p,i}) \Rightarrow R_i^{[l,1]}(w_{p,i}) \rightarrow R_i^{[l]}(c[p]) \rightarrow R_i^{[l,2]}(w_{p,i}) \Rightarrow W_p^{[k_{j'''}]}(w_{p,i}) \rightarrow W_p^{[k_j]}(c[p])$. This implies $R_i^{[l]}(c[p]) \rightarrow W_p^{[k_j]}(c[p])$, contradicting the assumption that $\pi(R_i^{[l]}(c[p]))$ is $W_p^{[k_j]}(c[p])$. The assertion follows.

(b and c) Let $label[p, x']$ be the variable in which $TW_p^{[k_j]}$ writes.

For j' described in part (a), we have $R_i^{[l]}(label[p, x]) \rightarrow R_i^{[l,2]}(w_{p,i}) \Rightarrow W_p^{[k_{j'}]}(w_{p,i}) \rightarrow TW_p^{[k_{j+2}]}$. That is, $TR_{i,p}^{[l]}$ finishes reading $label[p, x]$ before the traceable Write $TW_p^{[k_{j+2}]}$ starts its execution. From (i) the assumption that $\pi(R_i^{[l]}(c[p]))$ is $W_p^{[k_j]}(c[p])$, (ii) the property that $TW_p^{[k_{j+1}]}$ does not write in the same main label variable that $TW_p^{[k_j]}$ writes, (iii) $W_p^{[k_j]}(label[p, x']) \rightarrow W_p^{[k_j]}(c[p]) \Rightarrow R_i^{[l]}(c[p]) \rightarrow R_i^{[l]}(label[p, x])$, and (iv) Statements 1–3 in TRACEABLE-WRITE, it follows that $x = x'$, and $TW_p^{[k_j]}$ finishes writing $label[p, x]$ before $TR_{i,p}^{[l]}$ starts reading it. The assertions follow. □

LEMMA 6. *Let $TR_{i,p}^{[l]}$ be a traceable Read that finds $r_{i,p} = w_{p,i}$ on $R_i^{[l,2]}(w_{p,i})$. Suppose $TW_p^{[k_j]}$ is the traceable Write such that $R_i^{[l,1]}(w_{p,i}) \Rightarrow W_p^{[k_j]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i})$.*

(*a*) *The traceable Read $TR_{i,p}^{[l]}$ reading $copylabel[p, i]$ does not conflict with any traceable Write writing it.*

(*b*) $\pi(TR_{i,p}^{[l]}) = TW_p^{[k_j]}$.

PROOF

(a and b) By Lemma 4, $TW_p^{[k_j]}$ is the only traceable Write such that $R_i^{[l,1]}(w_{p,i}) \Rightarrow W_p^{[k_j]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i})$. It is clear from the TRACEABLE-WRITE procedure that $TW_p^{[k_j]}$ writes the value in $copylabel[p, i]$ (Statement 4.2.1) before setting the $w_{p,i}$ and $r_{i,p}$ values equal (Statement 4.2.3). This equality will not be changed until P_i starts the next traceable Read. Thus, the traceable Write $TW_p^{[k_{j+1}]}$ and subsequent traceable Writes of P_p, if they find $r_{i,p} = w_{p,i}$, will not write the copy label variable. From $W_p^{[k_j]}(copylabel[p, i]) \rightarrow W_p^{[k_j]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i}) \rightarrow R_i^{[l]}(copylabel[p, i])$, we have $W_p^{[k_j]}(copylabel[p, i]) \rightarrow R_i^{[l]}(copylabel[p, i])$. The assertions follow. □

Now we would like to show that private values of processes P_p are traceable. If a process P_i in its current label uses a private value v of another process P_p, P_i informs this "using of" v by setting $lend[i, p][1][i]$ to v at the end of the corresponding traceable Write (Statements 5–6). Thus, all the private values in the existing labels are traceable by their respective owners. The following lemma shows that the private values used by Scans are also traceable.

LEMMA 7. *Let a Scan $S_i^{[l]}$ of a process P_i use a private value v of a process P_p that has written the value v in a traceable Write $TW_p^{[k_j]}$. Then, P_p does not recycle v until $S_i^{[l]}$ is complete.*

PROOF. We need to consider the following two cases.

Case 1. $S_i^{[l]}$ got v directly from P_p.

We need to consider two subcases.

Subcase a. If the traceable Read $TR_{i,p}^{[l]}$ returns the value v from $copylabel[p, i]$, then, by Lemmas 6 and 4, the traceable Write $TW_p^{[k_j]}$ has executed the *if*-statement body (Statement 4.2) for process P_i. There it has set $myLend_p[p][1][i]$ to v

(Statement 4.2.2). The successive traceable Writes of P_p that occur before $S_i^{[l]}$ is complete will not execute the *if*-statement, and hence, will not change the $myLend_p[p][1][i]$ value. (Statement 5 does not change the value too.) As the labeling operation executions of P_p do not reuse the values referred to in $lend[1..n, p]$, v will not be reissued at least until $S_i^{[l]}$ is complete (Statements 1–2 in LABELING).

Subcase b. If the traceable Read $TR_{i,p}^{[l]}$ returns the value v from a main label variable, then by Lemma 5(a), traceable Write $TW_p^{[k_j]}$ or $TW_p^{[k_{j+1}]}$ executes the *if*-statement for process P_i. In the case of $TW_p^{[k_j]}$, $myLend_p[p][1][i]$ is set to v, and in the case of $TW_p^{[k_{j+1}]}$, $myLend_p[p][0][i]$ is set to v (Statements 4.2.2 and 7). The successive traceable Writes of P_p that occur before $S_i^{[l]}$ is complete will not execute the *if*-statement, and hence, will not change the $myLend_p[p][0..1][i]$ values. (Statement 5 does not change the values too.) By Lemma 1, $TW_p^{[k_{j+1}]}$ uses a private value different from v. So, by the argument given in the Subcase a, v will not be reissued as a new private value until $S_i^{[l]}$ is complete.

Case 2. $S_i^{[l]}$ got v from another process P_q.

CLAIM 1. *Process P_q has obtained v directly from P_p.*

PROOF. Note $S_i^{[l]}$ got v by reading a label from P_q. That is, P_q writes v in the pth component of the label. To form a new label, P_q uses the jth component of the labels it reads from processes P_j (Statements 5–6 in LABELING). Hence, P_q obtains v directly from P_p. □

Let $L_q^{[m_o]}$ be the corresponding labeling operation execution. Note that each labeling operation execution also executes traceable Reads (Statement 5). Then $\pi(TR_{q,p}^{[m_o]})$ is $TW_p^{[k_j]}$ and $\pi(TR_{i,q}^{[l]})$ is $TW_q^{[m_o]}$. As argued in Case 1, either $TW_p^{[k_j]}$ or $TW_p^{[k_{j+1}]}$ stores v in $myLend_p[p][0..1][q]$. This value will not be changed until $L_q^{[m_o]}$ is complete, in fact until P_q starts its next operation execution $O_q^{[m_o+1]}$. Let $TW_p^{[k_{j'}]}$, $j' \geq j+1$, be the first traceable Write that changes the $myLend_p[p][0..1][q]$ values different from v. Then, it must have found $L_q^{[m_o]}$ is complete and the next operation execution of P_q, namely $O_q^{[m_o+1]}$, has started. From $W_q^{[m_o]}(lend[q, p]) \rightarrow O_q^{[m_o+1]}(\mathcal{O}) \dashrightarrow L_p^{[k_{j'}]}(\mathcal{O}) \rightarrow L_p^{[k_{j'+1}]}$, we have $W_q^{[m_o]}(lend[q, p]) \rightarrow L_p^{[k_{j'+1}]}$. That is, $L_p^{[k_{j'+1}]}$ and successive labeling operation executions of P_p would not reissue v if v is found in $lend[q, p]$ (Statements 1–2). Note that $TW_q^{[m_o]}$ will write v in $lend[q, p][1][q]$ at the end of its execution (Statements 5–6 in TRACEABLE-WRITE). Also note that the traceable Write $TW_p^{[k_{j'}]}$ (actually $L_p^{[k_{j'}]}$) does not issue v. Now, from $\pi(TR_{i,q}^{[l]})$ is $TW_q^{[m_o]}$ it follows, by Lemmas 5 and 6, that either $TW_q^{[m_o]}$ or $TW_q^{[m_{o+1}]}$ would execute the *if*-statement for P_i, and write v in $myLend_q[p][0..1][i]$ indicating that the private value v of P_p is being used by P_i, and this will not be changed until $S_i^{[l]}$ is complete; in fact, until the next operation execution $O_i^{[l+1]}$ of P_i starts. Hence, $L_p^{[k_{j'+1}]}$ and successive labeling operation executions of P_p that may occur before $S_i^{[l]}$ is complete are able to trace v in $lend[q, p]$, and hence, will not reissue v. □

COROLLARY 1. *It is clear from the proof of Lemma 7 that if a Scan* $S_i^{[l]}$ *uses a private value* v *of* P_p *which is written in labeling operation execution* $L_p^{[k_j]}$, *then* $TW_p^{[k_j]}(\mathcal{O}[p])$ --▸ $TR_{i,p}^{[l]}(\mathcal{O}[p])$ *for direct reading and* $TW_p^{[k_j]}(\mathcal{O}[p])$ --▸ $TR_{q,p}^{[m_o]}(\mathcal{O}[p]) \rightarrow TW_q^{[m_o]}(\mathcal{O}[q])$ --▸ $TR_{i,q}^{[l]}(\mathcal{O}[q])$ *for indirect reading of* v *via process* P_q. *For the latter relation, by the axioms of Anger* [1989], $TW_p^{[k_j]}(\mathcal{O}[p])$ --▸ $TR_{i,q}^{[l]}(\mathcal{O}[q])$.

The following lemma shows that Scans can determine the correct temporal order of the private values of all processes.

LEMMA 8. *Let* $S_i^{[l]}$ *be a Scan that uses private values* v *and* v' *of a process* P_p. *Then,* $S_i^{[l]}$ *can determine the correct temporal order between the values* v *and* v'.

PROOF. Assume Scan $S_i^{[l]}$ uses the two different private values v and v' of process P_p that has written them in traceable Writes $TW_p^{[k_j]}$ and $TW_p^{[k_{j'}]}$, respectively, where $j < j'$, and hence, $v \prec_p v'$ (as defined in Section 3). By Lemma 7, P_p does not recycle v and v' until $S_i^{[l]}$ is complete. To guarantee the correctness of the timestamp system, we need to make sure that $S_i^{[l]}$ can correctly determine the order $v \prec_p v'$ in case these values are used in ordering some of the scanned labels. From the LABELING and SCAN routines and Corollary 1, we have $W_p^{[k_{j'}]}(order[p,i]) \rightarrow TW_p^{[k_{j'}]}(\mathcal{O}[p])$ --▸ $TR_{i,q}^{[l]}(\mathcal{O}[q]) \rightarrow R_i^{[l]}(order[p,i])$, where q is as defined in Corollary 1. That is, $W_p^{[k_{j'}]}(order[p,i]) \rightarrow R_i^{[l]}(order[p,i])$. Now, we need to make sure that $L_p^{[k_{j'}]}$ can correctly determine that the private value v is being used by the process P_i, before writing $order[p,i]$. Of course, it would assume v' could be used by P_i too. Since it knows $v \prec_p v'$, to inform this ordering to P_i, it writes v at a lower indexed entry in $order[p,i]$ than v'. The successive labeling operation executions do not change this ordering. Thus, P_i can determine the order of v and v' correctly after reading $order[p,i]$, by the regularity of order variables.

Now we answer the question how $L_p^{[k_{j'}]}$ finds that v might be used by P_i. Note that P_p does not know precisely which of its private values P_i is going to use. So, it guesses a subset of its private values, which contains the values actually being used by P_i. There are two cases to be considered.

Case 1. P_i obtains v directly from P_p. Either $TW_p^{[k_j]}$ or $TW_p^{[k_{j+1}]}$ will reserve v for P_i by storing v in $lend[p,p][0..1][i]$, and hence the use of v by P_i is traceable.

Case 2. P_i obtains v indirectly through another process P_q, for some q. From the claim in the proof of Lemma 7, we know that P_q has obtained v directly from P_p. Let the corresponding labeling operation execution be $L_q^{[m_0]}$. Either $TW_p^{[k_j]}$ or $TW_p^{[k_{j+1}]}$ will set $lend[p,p][0..1][q]$ to v, and P_p assumes v could be used by any process P_i through $\mathcal{O}[q]$ (one level of indirect propagation of a private value). At the end of $L_q^{[m_o]}$, in $TW_q^{[m_o]}$, P_q informs P_p that v is in $\mathcal{O}[q]$ by setting $lend[q,p][1][q]$ to v (Statements 5–6), and this value could be used by any process P_i. Alternatively, if P_q detects that the v is being used by P_i, it informs "this using" through $lend[q,p][0..1][i]$ (Statements 4.2.2 and 6).

Hence, if $L_p^{[k_{j'}]}$ finds v in $lend[p,p][0..1][i]$ or $lend[p,p][0..1][q]$ or $lend[q,p][1][q]$ or $lend[q,p][0..1][i]$, for some q, it will assume that v is being used by P_i (Statements 1 and 4.1 in LABELING procedure).

The assertion follows. □

CLAIM 2. *Each order variable is of size at most* $5n$.

PROOF. As discussed in the proof of Lemma 8, P_p needs to reserve its private values referred to in $lend[q, p][0..1][i]$, $lend[q, p][1][q]$ and $lend[p, p][0..1][q]$ for all q, that is, at most $5n$ values for process P_i. The claim follows. □

COROLLARY 2. *The set of private values is bounded. In fact, by Statements* 1–2 *in the LABELING procedure, the size of the set is less than* $2n^2$.

By the discussion at the end of the 3rd paragraph of Section 3, the correctness of the proposed construction is immediate. However, for the sake of completeness, we give the proof in Theorem 1. Before that, a technical lemma follows.

LEMMA 9. *Let* $TR_{i,p}^{[l]}$ *and* $TR_{i',p}^{[l']}$ *be two traceable Reads such that* $TR_{i,p}^{[l]} \rightarrow TR_{i',p}^{[l']}$ *and* $\pi(TR_{i,p}^{[l]})$ *be* $TW_p^{[k_j]}$. *Then,*

(*a*) $W_p^{[k_j]}(c[p]) \Rightarrow R_{i'}^{[l']}(c[p])$,
(*b*) $\pi(TR_{i',p}^{[l']})$ *is* $TW_p^{[k_{j'}]}$, *where* $j' \geq j$, $k_{j'} \geq k_j$.

PROOF. We have the following two cases.

Case 1. $TR_{i,p}^{[l]}$ finds $r_{i,p} \neq w_{p,i}$ on $R_i^{[l,2]}(w_{p,i})$.

Lemma 5 (b) implies that $\pi(R_i^{[l]}(c[p]))$ is $W_p^{[k_j]}(c[p])$. Then, we have $TW_p^{[k_{j-1}]} \rightarrow W_p^{[k_j]}(c[p]) \Rightarrow R_i^{[l]}(c[p]) \rightarrow R_{i'}^{[l',1]}(w_{p,i'}) \rightarrow R_{i'}^{[l']}(c[p])$.

Case 2. $TR_{i,p}^{[l]}$ finds $r_{i,p} = w_{p,i}$ on $R_i^{[l,2]}(w_{p,i})$.

By Lemma 6, we have $TW_p^{[k_{j-1}]} \rightarrow W_p^{[k_j]}(c[p]) \rightarrow W_p^{[k_j]}(w_{p,i}) \Rightarrow R_i^{[l,2]}(w_{p,i}) \rightarrow R_{i'}^{[l',1]}(w_{p,i'}) \rightarrow R_{i',p}^{[l']}(c[p])$.

For both cases, we have $W_p^{[k_j]}(c[p]) \Rightarrow R_{i'}^{[l']}(c[p])$; part (a) follows. If $TR_{i',p}^{[l']}$ finds $r_{i',p} \neq w_{p,i'}$ on $R_{i'}^{[l',2]}(w_{p,i'})$, then part (b) follows by Lemma 5. Assume $TR_{i',p}^{[l']}$ finds $r_{i',p} = w_{p,i'}$ on $R_{i'}^{[l',2]}(w_{p,i'})$. From the above two cases, we have $TW_p^{[k_{j-1}]} \rightarrow R_{i'}^{[l',1]}(w_{p,i'})$. Then part (b) follows by Lemmas 4 and 6. □

THEOREM 1. *The construction of Figure* 1 *is a correct implementation of wait-free bounded concurrent timestamp systems.*

PROOF. The wait-freedom property is immediate from the structure of the four routines in Figure 1. The boundedness follows from Corollary 2. We now show that the construction satisfies all the four properties P1–P4 described in Section 2.

Ordering. Consider two labeling operation executions $L_p^{[k]}$ and $L_q^{[k']}$ with labels $l_p^{[k]}$ and $l_q^{[k']}$, respectively. Let m be the least significant index such that $l_p^{[k]}[m] \neq l_q^{[k']}[m]$. Assume these private values $l_p^{[k]}[m]$ and $l_q^{[k']}[m]$ are written by P_m at labeling operation executions $L_m^{[s_o]}$ and $L_m^{[s_{o'}]}$, respectively. We define $L_p^{[k]} \Rightarrow L_q^{[k']}$ iff $L_m^{[s_o]} \rightarrow L_m^{[s_{o'}]}$.

—*Precedence*. Without loss of generality, we assume $L_p^{[k]} \rightarrow L_q^{[k']}$. By Lemmas 5 and 6, we have $\pi(TR_{p,m}^{[k]})$ is $TW_m^{[s_o]}$ and $\pi(TR_{q,m}^{[k']})$ is $TW_m^{[s_{o'}]}$. Then, from $TR_{p,m}^{[k]} \rightarrow TR_{q,m}^{[k']}$ and Lemma 9(b), we have $s_{o'} \geq s_o$. As $l_p^{[k]}[m] \neq l_q^{[k']}[m]$, we have $s_{o'} \neq s_o$, and hence, $s_{o'} > s_o$. That is, $L_m^{[s_o]} \rightarrow L_m^{[s_{o'}]}$. The precedence property follows.

—*Consistency*. For any two labels $l_p^{[k]}$ and $l_q^{[k']}$ (returned by a Scan) such that m is the least significant index for which $l_p^{[k]}[m] \neq l_q^{[k']}[m]$. We define $l_p^{[k]} \prec l_q^{[k']}$ iff $l_p^{[k]}[m] \prec_m l_q^{[k']}[m]$ iff $L_m^{[s_o]} \rightarrow L_m^{[s_{o'}]}$. The consistency property follows by Lemma 8 and the definition of $\Rightarrow$ given above.

Regularity. Consider a Scan $S_i^{[j]}$ that returns a label $l_p^{[m_o]}$ that is written by a labeling operation execution $L_p^{[m_o]}$, that is, $\pi(TR_{i,p}^{[j]})$ is $TW_p^{[m_o]}$. By Lemmas 5 and 6, we can say $TW_p^{[m_o]} \dashrightarrow TR_{i,p}^{[j]}$, and hence, $L_p^{[m_o]} \dashrightarrow S_i^{[j]}$. The second part of the regularity property follows from: (i) if $TR_{i,p}^{[j]}$ finds $r_{i,p} \neq w_{p,i}$ on $R_i^{[j,2]}(w_{p,i})$, then, by Lemma 5, $\pi(TR_{i,p}^{[j]})$ is $TW_p^{[m_o]}$, where $\pi(R_i^{[j]}(c[p]))$ is $W_p^{[m_o]}(c[p])$, and so, $TW_p^{[m_{o+1}]} \not\rightarrow TR_{i,p}^{[j]}$, and hence $L_p^{[m_{o+1}]} \not\rightarrow S_i^{[j]}$; (ii) if $TR_{i,p}^{[j]}$ finds $r_{i,p} = w_{p,i}$ on $R_i^{[j,2]}(w_{p,i})$, then, by Lemma 6, $\pi(TR_{i,p}^{[j]})$ is $TW_p^{[m_o]}$, where $R_i^{[j,1]}(w_{p,i}) \Rightarrow W_p^{[m_o]}(w_{p,i}) \Rightarrow R_i^{[j,2]}(w_{p,i})$, and so, $TW_p^{[m_{o+1}]} \not\rightarrow TR_{i,p}^{[j]}$, and hence $L_p^{[m_{o+1}]} \not\rightarrow S_i^{[j]}$.

Monotonicity. Consider two Scans $S_i^{[j]} \rightarrow S_{i'}^{[j']}$. Let $S_i^{[j]}$ return label $l_p^{[m_o]}$ from a process P_p. By Lemmas 5 and 6, we have $\pi(TR_{i,p}^{[j]})$ is $TW_p^{[m_o]}$. From $S_i^{[j]} \rightarrow S_{i'}^{[j']}$, we have $TR_{i,p}^{[j]} \rightarrow TR_{i',p}^{[j']}$. The monotonicity property follows by Lemma 9.

Extended Regularity. Consider a Scan $S_i^{[j]}$ that returns a label $l_p^{[m_o]}$ that is written by a labeling operation execution $L_p^{[m_o]}$, that is, $\pi(TR_{i,p}^{[j]})$ is $TW_p^{[m_o]}$. For each labeling operation execution $L_q^{[m']}$, if $S_i^{[j]} \rightarrow L_q^{[m']}$, then $TR_{i,p}^{[j]} \rightarrow TR_{q,p}^{[m']}$. Then, by Lemma 9(a), we have $W_p^{[m_o]}(c[p]) \Rightarrow R_q^{[m']}(c[p])$ and hence, $\pi(TR_{q,p}^{[m']})$ is $TW_p^{[m_o]}$ or a successor, by Lemma 9(b). Also by Lemmas 5 and 6 and the LABELING procedure, we have $TR_{p,s}^{[m_o]} \rightarrow TW_p^{[m_o]} \dashrightarrow TR_{i,p}^{[j]} \rightarrow TR_{q,s}^{[m']}$ for all $s \neq p$, that is, $TR_{p,s}^{[m_o]} \rightarrow TR_{q,s}^{[m']}$. Hence, $L_q^{[m']}$ reads more recent (at least equal) private values of all processes than $L_p^{[m_o]}$. Also, we have $l_p^{[m_o]}[q] \prec_q l_q^{[m']}[q]$. Hence, $L_p^{[m_o]} \Rightarrow L_q^{[m']}$. The extended regularity property follows. □

5. *Concluding Remarks*

This article combines the preliminary articles [Vitányi and Awerbuch 1986; Haldar 1993]. The former article is the first to characterize multiwriter shared variables, and provides a bounded construction of the multiwriter-multireader-multivalued atomic variable from 1-writer variables. However, it was later found that the proposed construction doesn't satisfy some properties of atomic shared variables [Vitányi and Awerbuch 1987]. The technical report [Haldar 1993] corrected and extended [Vitányi and Awerbuch 1986] to a construction of a concurrent timestamp

TABLE I. COMPARISON RESULTS

Construction	Shared variable size	Shared space (bits)	Labeling	Scan
Dolev and Shavit [1989/1997]	$O(n)$	$O(n^3)$	$O(n)$	$O(n^2 \log n)$
Gawlick et al. [1992]	$O(n^2)$	$O(n^4)$	$O(n \log n)$	$O(n \log n)$
Israeli and Pinhasov [1992]	$O(n^2)$	$O(n^4)$	$O(n)$	$O(n)$
Dwork and Waarts [1992/1999]	$O(n \log n)$	$O(n^5 \log n)$	$O(n)$	$O(n)$
Dwork et al. [1992/1999]	$O(n)$	$O(n^3)$	$O(n)$	$O(n)$
This article	$O(n \log n)$	$O(n^3 \log n)$	$O(n)$	$O(n)$

system using an idea from Dwork and Waarts [1992/1999]. The final result is very close to the incorrect construction of Vitányi and Awerbuch [1986]. It uses $O(n \log n)$ bit-size shared variables (*order* and *lend* variables), where n is the number of processes. Scan and labeling operation executions require $O(n)$ steps. The construction uses less shared space than that of Dwork and Waarts [1992/1999] at the fundamental level, and is orders-of-magnitude more efficient in terms of scanning bits at the fundamental level.

5.1. COMPARISON WITH RELATED WORK. In Dwork and Waarts [1992/1999], they have defined three routines, namely, traceable-read, traceable-write and garbage collection. When the traceable-read function is executed to read a label, the executing process explicitly informs the other processes which of their private values it is going to use. The traceable-write procedure is executed to write a new label. To determine which of its private values are currently in use, a process executes the garbage collection routine. This routine helps processes to safely recycle their respective private values. This is the most intricate routine.

In our construction, we have used a separate implementation technique for a weaker form of the traceable-read and the traceable-write routines. We do not need a garbage collection routine. When a process executes the traceable-read function, it does not explicitly inform the other processes which of their private values it is going to use. On the other hand, the executers of the traceable-write procedure correctly finds which private values of which processes are in use.

Every process needs a separate pool of private values, whose size is fewer than $2n^2$. In their construction, the pool size is $22n^2$. All the ordering shared variables used in our construction are of 1-writer 1-reader regular ones, whereas they use 1-writer n-reader atomic ones in their construction. In our construction, a Scan reads at most $n - 1$ 1-writer 1-reader regular order shared variables, whereas in their construction it is $2n - 2$ 1-writer n-reader atomic ones. In our construction, all but one bit are nonatomic 1-writer 1-reader variables. Table I presents some comparison results briefly.

Of all proposed constructions of bounded concurrent timestamp systems we are aware of, the construction in this article is the "simplest." The correctness proof, though involved, is easier to follow. It is used as a basis in the reference text [Attiya and Welch 1998] to describe bounded concurrent timestamp system.

Although we have used a notion of vector clocks for our construction, as in Vitányi and Awerbuch [1986], we may not really need the full power of vector clock concept developed later by Mattern [1989]. In CTSs, we are not interested in determining causal "independence" of various labeling operation executions. The ordering property of CTSs implies that the causal orders among labeling operation executions matter most. We need to have a total order on all labeling operation

executions, and the total order must extend their original causal relation. This is akin to the logical time of Lamport [1978]. We suspect that there might be a way to eliminate the vector clock altogether, by an efficient way of recycling of global values, instead of using n sets of private values.

The construction presented here should not be considered as an alternative implementation of the traceable use abstraction, for it restricts the value propagation at indirection level one. It is not clear to the authors how this strategy could be extended for a general implementation of the abstraction.

5.2. A BRIEF EARLY HISTORY. The development of bounded wait-free shared variables and timestamp systems has been quite problematic and error-prone. It may be useful at this point to present a brief early history of the area: who did what, when, and where, and which solutions are known to be incorrect. In a series of articles starting in 1974, Lamport [1974, 1977, 1978, 1986] explored various notions of concurrent reading and writing of shared variables culminating in the seminal 1986 paper [Lamport 1986]. It formulates the notion of wait-free implementation of an atomic shared variable—written by a single writer and read by (another) single reader—from safe 1-writer 1-reader 2-valued shared variables, being mathematical versions of physical *flip-flops*. Predating the latter article, Peterson [1983] published an ingenious wait-free construction of an atomic 1-writer, n-reader m-valued atomic shared variable from $n+2$ safe 1-writer n-reader m-valued registers, $2n$ 1-writer 1-reader 2-valued atomic shared variables, and two 1-writer n-reader 2-valued atomic shared variables. He presented also a proper notion of wait-freedom property. Lamport [1984] gave an example that appeared to contradict a possible interpretation of the informal statement of a theorem in Peterson [1983], which, as Peterson apparently retorted to Lamport, was not intended. In his paper, Peterson didn't tell how to construct the n-reader Boolean atomic variables from flip-flops, while Lamport mentioned the open problem of doing so, and, incidentally, uses a version of Peterson's construction to bridge the algorithmically demanding step from atomic shared bits to atomic shared multivalues. Based on this work, N. Lynch, motivated by concurrency control of multiuser databases, posed around 1985 the question of how to construct wait-free multiwriter atomic variables from 1-writer multireader atomic variables (personal knowledge of the author PV). Her student Bloom [1987/1988] found in 1986 an elegant 2-writer construction, which, however, has resisted generalization to multiwriter. Vitányi and Awerbuch [1986] were the first to define and explore the complicated notion of wait-free constructions of general multiwriter atomic variables. They presented a proof method, an unbounded solution from 1-writer 1-reader atomic variables, and a bounded solution from 1-writer n-reader atomic variables. The unbounded solution was made bounded in Li et al. [1987/1996]. It is optimal for the implementation of n-writer n-reader atomic variables from 1-writer 1-reader ones. "Projections" of the construction also give specialized constructions for the implementation of 1-writer n-reader atomic variables from 1-writer 1-reader ones, and for the implementation of n-writer n-reader atomic variables from 1-writer n-reader ones. As noted in Li and Vitányi [1992], the first "projection" is optimal, while the last "projection" may not be optimal since it uses $O(n)$ control bits per writer while only a lower bound of $\Omega(\log n)$ was established. Taking up this challenge, the construction in Israeli and Shaham [1992] apparently achieves this lower bound. The earlier bounded solution in Vitányi and Awerbuch [1986] (corresponding in fact to

the problem correctly solved by the last "projection" above) turned out not to be atomic, but only achieved regularity [Vitányi and Awerbuch 1987]. Nonetheless, Vitányi and Awerbuch [1986] introduced important notions and techniques in the area, like (bounded) vector clocks. These were inspired by the celebrated "Bakery" algorithm of Lamport [1974], which can be viewed as a global bounded "clock" determining the order among queued processes much like the ticket dispenser in a bakery serves to determine the order of servicing waiting customers. The multiwriter situation has stronger requirements than apparently can be satisfied by a global ticket dispenser. The solution in Vitányi and Awerbuch [1986] was the construction of a bounded "vector clock": a private ticket dispenser for each process, the storing and updating of a vector of latest tickets held by all processes, together with a semantics to determine the order between vectors. Moreover, a complex mechanism–primitive traceable read/write—is presented to keep track of which tickets of what processes could still be present in the system, with the objective of bounding the private ticket pool of each process by recycling obsolete tickets. Following the appearance of Vitányi and Awerbuch [1986], Peterson, who had been working on the multiwriter problem for a decade, together with Burns, revamped the construction retaining the vector clocks, but replaced the primitive traceable read/write elements by repeated scanning as in Peterson [1983]. The result [Peterson and Burns 1987] was found to be nonetheless erroneous, in the technical report [Schaffer 1988]. This makes the multiwriter problem perhaps the only one for which two consecutive wrong solutions were published in the highly selective FOCS conferences. Neither the recorrection in Schaffer [1988], nor the claimed recorrection by [Peterson and Burns 1987] has appeared in print. The present article constitutes a correction of the original [Vitányi and Awerbuch 1986] by the extension of Haldar [1993]: by implementing the stronger concurrent timestamp system, it also solves the atomic multiwriter problem. Apart from the already-mentioned article [Li et al. 1987/1996], the only other multiwriter-multireader atomic shared variable construction that appeared in journal version seems to be of Abraham [1995]. Also, in 1987, there appeared at least five purported solutions for the implementation of 1-writer n-reader atomic shared variable from 1-writer 1-reader ones: Kirousis et al. [1987], Newman-Wolfe [1987], Burns and Peterson [1987], and Singh et al. [1987/1994] and the conference version of Israeli and Li [1987/1993], of which Burns and Peterson [1987] was shown to be incorrect in Haldar and Vidyasankar [1992], and only Singh et al. [1987/1994] appeared in journal version. The only other 1-writer n-reader atomic shared variable construction that appeared in journal version is of Haldar and Vidyasankar [1995a]. Israeli and Li were attracted to the area by the work in Vitányi and Awerbuch [1986] and, in an important paper [Israeli and Li 1987/1993], they raised and solved the question of the more general and universally useful notion of bounded timestamp system to track the order of events in a concurrent system. Their sequential timestamp system was published in journal version, but the preliminary concurrent timestamp system in the conference proceedings, of which a more detailed version has been circulated in manuscript form, has not been published in final form.

The difficulty of wait-free atomic multireader-, multiwriter-, and timestamp-system constructions, and the many errors in purported and published solutions, have made it hard to publish results in print. Of the major pioneering papers, the first correct multiwriter construction of 1987 [Li et al. 1987/1996] was rejected at five consecutive conferences until it was published in ICALP, 1989.

The final journal version was handled by three consecutive editors, scrutinized by three consecutive sets of referees, and lasted from 1989 until publication in 1996. The pioneering timestamp paper [Israeli and Li 1987/1993] was submitted in 1987/88 to this journal, after a couple of years refereeing, surprisingly rejected since a stronger result [Dolev and Shavit 1989/1997] had appeared in conference version, submitted to another journal and finally appeared in 1993, but only the part containing the simpler sequential timestamp construction. The first generally accepted concurrent timestamp construction [Dolev and Shavit 1998/1997] appeared in conference version in 1989, but its journal version appeared in 1997. As stated before, the concurrent timestamp construction in the present article is based on the 1986 article [Vitányi and Awerbuch 1986], supplemented by the 1993 technical report [Haldar 1993]. For further remarks, see Li et al. [1987/1996] in this journal and the Introduction to present article.

ACKNOWLEDGMENT. Hagit Attiya and Orli Waarts gave valuable suggestions for an early version of Haldar [1993], and Baruch Awerbuch co-authored the preliminary article [Vitányi and Awerbuch 1986] on which this article is based.

REFERENCES

ABRAHAM, U. 1995. On interprocess communication and the implementation of multi-writer atomic registers. *Theoret. Comput. Sci. 149*, 2, 257–298.

ABRAHAMSON, K. 1988. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 291–302.

ANGER, F. 1989. On Lamport's interprocess communication model. *ACM Trans. Prog. Lang. Syst. 11*, 3, 404–417.

ATTIYA, H., AND WELCH, J. 1998. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, London, UK.

BLOOM, B. 1988. Constructing two-writer atomic registers. *IEEE Trans. Comput. 37*, 12, 1506–1514. (Preliminary version: Constructing two-writer atomic registers. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 249–259, 1987.)

BURNS, J. E., AND PETERSON, G. L. 1987. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 222–231.

CHOR, B., ISRAELI, A., AND LI, M. 1987. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 86–97.

DIJKSTRA, E. W. 1965. Solutions of a problem in concurrent programming control. *Commun. ACM 8*, 9, 165–165.

DOLEV, D., AND SHAVIT, N. 1997. Bounded concurrent time-stamp systems are constructible. *SIAM J. Comput. 26*, 2, 418–455. (Preliminary version in: *Proceedings of the 21st ACM Symposium on Theory of Computing*. ACM, New York, pp. 454–466, 1989.)

DWORK, C., HERLIHY, M., PLOTKIN, S., AND WAARTS, O. 1992. Time-lapse snapshots. In *Proceedings of Israeli Symposium on Theory of Computing and Systems*. Lecture Notes in Computer Science, vol. 601. Springer-Verlag, New York, pp. 154, 170. (Also, in *SIAM J. Comput. 28*, 5, 1848–1874, 1999.)

DWORK, C., AND WAARTS, O. 1999. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. ACM 46*, 5, 633–666. (Preliminary version in: *Proceedings of the 24th ACM Symposium on Theory of Computing*. ACM, New York, pp. 655–666, 1992.)

FISCHER, M. J., LYNCH, N. A., BURNS, J. E., AND BORODIN, A. 1979. Resource allocation with immunity to limited process failure. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif. pp. 234–254.

FISHBURN, P. C. 1985. *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets*. Wiley, New York.

GAWLICK, R., LYNCH, N. A., AND SHAVIT, N. 1992. Concurrent timestamping made simple. In *Proceedings of Israeli Symposium on Theory of Computing and Systems*. Lecture Notes in Computer Science, vol. 601. Springer-Verlag, New York, pp. 171–183.

HALDAR, S. 1993. Efficient Bounded Timestamping Using Traceable Use Abstraction—Is Writer's Guessing Better Than Reader's Telling? Tech. Rep. RUU-CS-93-28, Dept. of Computer Science, Utrecht University, The Netherlands.

HALDAR, S., AND VIDYASANKAR, K. 1992. Counterexamples to a one writer multireader atomic shared variable construction of Burns and Peterson. *ACM Oper. Syst. Rev 26*, 1, 87–88.

HALDAR, S., AND VIDYASANKAR, K. 1995a. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM 42*, 1, 186–203.

HALDAR, S., AND VIDYASANKAR, K. 1995b. Buffer-optimal constructions of 1-writer multireader multivalued atomic shared variables. *J. Parall. Dist. Comput. 31*, 2, 174–180. (Preliminary version in: Conflict-free constructions of 1-writer multireader multivalued atomic shared variables. TR 9116, Dept. of Computer Science, Memorial University of Newfoundland, Canada, 1991.)

HALDAR, S., AND VIDYASANKAR, K. 1996. Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *ACTA Inf. 33*, 2, 177–202.

HERLIHY, M., AND WING, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3, 463–492.

ISRAELI, A., AND LI, M. 1993. Bounded time-stamps. *Dist. Comput. 6*, 205–209. (Preliminary version in: In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 371–382, 1987.)

ISRAELI, A., AND PINHASOV, M. 1992. A concurrent time-stamp scheme which is linear in time and space. In *Proceedings of the Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, Springer-Verlag, vol. 647, Berlin, pp. 95–109.

ISRAELI, A., AND SHAHAM, A. 1992. Optimal multi-writer multireader atomic register. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 71–82.

KIROUSIS, L. M., KRANAKIS, E., AND VITÁNYI, P. M. B. 1987. Atomic multireader register. In *Proceedings of the Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, vol, 312. Springer-Verlag, Berlin, pp. 278–296.

LAMPORT, L. 1974. A new solution to Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (Aug.), 453–455.

LAMPORT, L. 1977. On concurrent reading and writing. *Commun. ACM 20*, 11 (Nov.), 806–811.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LAMPORT, L. 1984. On a "Theorem" of Peterson Unpublished (October, 1984). http:// www.research. compaq.com/SRC/personal/lamport/pubs/pubs.html#peterson-theorem.

LAMPORT, L. 1986. On interprocess communication—Part I: Basic formalism, Part II: Algorithms. *Dist. Comput. 1*, 2, 77–101.

LI, M., AND VITÁNYI, P. M. B., 1992. Optimality of wait-free atomic multiwriter variables. *Inf. Process. Lett. 43*, 2, 107–112.

LI, M., TROMP, J., AND VITÁNYI, P. M. B. 1996. How to share concurrent wait-free variables. *J. ACM 43*, 4, 723–746. (Preliminary version: Li, M. and Vitányi, P. M. B. 1987. A very simple construction for atomic multiwriter register, Tech. Rept. TR-01-87, Computer Science Dept., Harvard University, Nov.)

MATTERN, F. 1989. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*. North-Holland / Elsevier, Amsterdam, The Netherlands, pp. 215–226. (Reprinted in: Z. Yang and T. A. Marsland, Eds., *Global States and Time in Distributed Systems*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 123–133.)

MATTERN, F. 1992. On the relativistic structure of logical time in distributed systems. In *Datation et Controle des Executions Reparties, Bigre 78* (ISSN 0221-525), pp. 3–20.

NEWMAN-WOLFE, R. 1987. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, pp. 232–248.

PETERSON, G. L. 1983. Concurrent reading while writing. *ACM Trans. Prog. Lang. Syst. 5*, 1, 56–65.

PETERSON, G. L., AND BURNS, J. E. 1987. Concurrent reading while writing. II: The multiwriter case. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 383–392.

SCHAFFER, R. 1988. On the correctness of atomic multiwriter registers. Report MIT/LCS/TM-364. Massachusetts Institute of Technology, Cambridge, Mass., pp. 1–58.

SINGH, A. K., ANDERSON, J. H., AND GOUDA, M. G. 1994. The elusive atomic register. *J. ACM 41*, 2, 311–339. (Preliminary version in: *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 1987.)

TROMP, J. 1989. How to construct an atomic variable. In *Proceedings of the Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, vol. 392. Springer-Verlag, Berlin, pp. 292–302.

VIDYASANKAR, K. 1990. Concurrent reading while writing revisited. *Dist. Comput. 4*, 81–85.

VIDYASANKAR, K. 1996. Weak atomicity: A helpful notion in the construction of atomic shared variables. *SADHANA: J. Eng. Sci. IAS 21*, 245–259.

VITÁNYI, P. M. B., AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 233–243.

VITÁNYI, P. M. B., AND AWERBUCH, B. 1987. Errata to "Atomic shared register access by asynchronous hardware". In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 487–487.

YAKOVLEV, A. 1993. Review of "Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible!" by C. Dwork and O. Waarts. *ACM Comput. Rev. 34*, 5, 260–261.

RECEIVED AUGUST 1999; REVISED OCTOBER 2001; ACCEPTED DECEMBER 2001