Efficient Bounded Timestamping Using Traceable Use Abstraction -

Is Writer's Guessing Better Than Reader's Telling?

S. Haldar

RUU-CS-93-28
September 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Efficient Bounded Timestamping Using Traceable Use Abstraction - Is Writer's Guessing Better Than Reader's Telling?

S. Haldar

Technical Report RUU-CS-93-28
September 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Efficient Bounded Timestamping Using Traceable Use Abstraction — Is Writer's Guessing Better Than Reader's Telling? *

S. Haldar
Department of Computer Science
Utrecht University , PO Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

Traceable use is a helpful abstraction to recycling values in bounded wait-free systems. Several researchers have demonstrated the power of the traceable use abstraction in constructing concurrent timestamping systems, snapshot variables, bounded round numbers. In this paper, we present an efficient implementation technique of the traceable use abstraction, which is finally used in developing a new construction of concurrent timestamping systems. This new construction is much simpler, in fact better, than the other traceable use abstraction based construction in the literature. The new implementation exhibits that sometimes writer's guessing is better than reader's explicit telling.

Index Terms: Concurrent reading while writing; label; operation — read and write, labeling and scan; operation execution; shared variable — safe, regular and atomic; timestamping system, traceable use, wait-freedom.

1 Introduction

Consider an asynchronous multiprocessing system consisting of a set of objects that are read and written by a set of processes. The system has no global clock or synchronization primitives.

*This research is partially supported by the Netherlands Organization for Scientific Research (NWO) under Contract Number NF 62-376 (NFI project ALADDIN: *Algorithmic Aspects of Parallel and Distributed Systems*).

Each object is associated with a process (called *owner*) which writes it and the other processes read it. One of the requirements of the system is to determine the temporal order of the objects in which they are written. For this purpose, each object is given a *label* (also refer to as *timestamp*) which indicates the latest (relative) time when it has been written by its owner process. The crucial role of the timestamps is to maintain the ordering of various writings of the objects. The processes label their respective objects in such a way that the object-labels reflect the real-time order in which they are written. Such system is known as *timestamping system* [14]. This system must support two operations, namely *labeling* and *scan*. A labeling operation execution (Labeling, in short) assigns a new label to an object, and a scan operation execution (Scan, in 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 temporarily. To construct a timestamping system, one needs additional shared space apart from the space for the objects.

In this paper, we are concerned with those systems where operations can be executed *concurrently*, i.e., in an overlapped fashion. Moreover, operation executions must be *wait-free*, that is, each operation execution will take at most a fixed amount of time (the number of accesses to shared space), irrespective of the presence of other operation executions and their relative speeds. We require the labeling operation executions whose intervals are disjoint to keep the right temporal order. On the other hand, we allow any ordering of labeling operation executions with overlapping intervals.

Constructing concurrent timestamping systems has been of much interest recently. It is a powerful tool for many concurrency control problems such as *fcfs*-mutual exclusion [5, 17], multiwriter multireader shared variables [26], probabilistic consensus [4, 1], *fcfs* l-exclusion [10], etc. We are interested in constructing concurrent timestamping systems using shared Read/Write variables (variables, in short) [18].

It is a trivial task to construct a timestamping system if the shared space is unbounded (i.e., there is no limit on the size of some shared variables). Here, we are particularly interested in bounded (shared space) systems. A *bounded timestamping system* is a timestamping system with a finite set of labels. Such system should reflect the temporal order among all existing objects, thereby the number of objects which may exist concurrently is bounded too. In the rest of the paper, unless specified otherwise, by a timestamping system we mean a wait-free bounded concurrent timestamping system.

The concept of bounded timestamps is introduced by Israeli and Li [14]. They also present a system in which operation executions are sequential. A construction of bounded concurrent timestamping system is first presented by Dolev and Shavit [6]. Their construction uses

shared variables of size $O(n)$, where n is the number of processes in the system. Each labeling operation execution requires $O(n)$ steps, and each Scan $O(n^2 \log n)$ steps. (A step is a read or write of a shared variable). Following Dolev and Shavit, several researchers have come out with different varieties of constructions. The construction of Israeli and Pinhasov [15] uses shared variables of size $O(n^2)$; labeling and scan operation executions require $O(n)$ steps. The construction of Dwork and Waarts [7] uses shared variables of size $O(n \log n)$; labeling and scan operation executions require $O(n)$ steps. The construction of Dwork, Herlihy, Plotkin and Waarts [8] uses shared variables of size $O(n)$; labeling and scan operation executions access $O(n)$ shared variables. The construction of Gawlik, Lynch and Shavit [11] uses shared variables of size $O(n^2)$; labeling and scan operation executions access $O(n^2)$ shared variables.

Among the constructions mentioned above, the one of Dwork and Waarts [7] is relatively simple and efficient too. In their paper, they have introduced a powerful concept called *traceable use abstraction* to recycling values of shared variables. Several researchers have demonstrated the usefulness of the traceable use abstraction by constructing timestamping systems [7, 8], atomic snapshot objects [2], bounded round numbers [9].

In this paper, we are interested in the constructions of concurrent timestamping systems from the view point of the traceable use abstraction. Dwork and Waarts in [7] have presented a technique to implement the traceable use abstraction, and have shown how it could be used in constructing timestamping systems. There, each label is a vector of n private values, one for each of n processes. 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 the other processes which of their private values it is going to use. To determine which of its private values are currently in use, a process executes a *garbage collection* routine. This routine helps processes to safely recycle their respective private values. Each process maintains a separate pool of (at least) $13n^2$ private values. As mentioned earlier, both labeling and scan operation execution access $O(n)$ shared variables. Truly speaking, the actual complexity of labeling operation executions is $O(n^2)$, because of the use of a complicated garbage collection routine that requires $O(n^2)$ steps. But, the amortized complexity is $O(n)$.

In this paper, we present an efficient implementation of the traceable use abstraction of Dwork and Waarts, which is oriented towards a new construction of concurrent timestamping systems. This construction is similar to that of Dwork and Waarts, but there are a lot of differences. We have used a separate implementation technique for traceable-read and 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 guess which private values of which processes are in use in the system. In the proposed construction, each local pool of private values contains fewer than $2n^2$ values. Above all, the proposed construction is much simpler than that of Dwork and Waarts, and a little efficient too. It also exhibits that sometimes writer's guessing is better than reader's telling.

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

2 Model, problem definition, and some notations

A concurrent system consists of a collection of asynchronous processes that communicate through a set of initialized data objects. A concurrent timestamping system is an abstract object V shared by a set of n asynchronous processes, $P_1, P_2, \ldots, P_n$. The object V has n components, $V[1..n]$. Component $V[p]$ is written by process P_p and read by all other processes. The system supports two activities, namely, writing new values in V, and determining the temporal order between any two components of V. For this purpose, a field (other than data value) called *label* or *timestamp* is associated with each component of V. When process P_p writes a new value from a well defined domain in $V[p]$, it also assigns a new label to $V[p]$. (From now onward, we will not consider the data values of the components of V.) The system supports two types of operations, namely, *labeling* and *scan*. A labeling operation execution by process P_p determines and assigns a new label to $V[p]$. It may use all existing labels of $V[1..n]$, but it is not allowed to change the labels of other components than $V[p]$. A scan operation execution returns a pair $(\bar{l}, \prec)$, where $\bar{l}$ is a set of current labels, one for each process, and $\prec$ is a total order on $\bar{l}$. Ordering among the subsets of labels returned by any Scan is in fact the same as the total ordering on all the labeling operation executions no matter how many labeling operation executions occurred while the labels were being scanned. Operation executions of each process are sequential. However, operation executions of different processes need not be sequential. Furthermore, we need operation executions to be *wait-free*, that is, each operation execution will take at most a fixed amount of time, irrespective of the presence of other operation executions and their relative speeds.

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

We say an operation execution A *precedes* another operation execution B, denoted $A \longrightarrow B$, if A finishes before B starts; A and B *overlap* if neither A precedes B nor B precedes A. 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 $A \longrightarrow B$ or A overlaps B; in other words, $B \not\longrightarrow A$. We also assume that if $B \not\longrightarrow A$, then $A \dashrightarrow B$. That is, we assume global time model [18].

A concurrent timestamping system must ensure the following properties [6, 7, 11].

P1. Ordering: There exists an irreflexive total order $\Rightarrow$ on the set of all labeling operation executions, such that the following two hold.

- Precedence: For any pair of labeling operation executions $L_p^{[k]}$ and $L_q^{[k']}$, if $L_p^{[k]} \longrightarrow L_q^{[k']}$ then $L_p^{[k]} \Rightarrow L_q^{[k']}$.
- Consistency: For any Scan $S_i^{[j]}$ returning $(\bar{l}, \prec)$, for any 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 any label $l_p^{[k]}$ in $\bar{l}$ returned by a Scan $S_i^{[j]}$, $L_p^{[k]}$ begins before $S_i^{[j]}$ terminates, i.e., $L_p^{[k]} \dashrightarrow S_i^{[j]}$, and there is no labeling operation execution $L_p^{[k']}$ such that $L_p^{[k]} \longrightarrow L_p^{[k']} \longrightarrow S_i^{[j]}$.

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 $S_i^{[j]} \longrightarrow S_{i'}^{[j']}$, then $k \leq k'$.

P4. Extended Regularity: For any label $l_p^{[k]}$ returned by a Scan $S_i^{[j]}$, if $S_i^{[j]} \longrightarrow L_q^{[k']}$ for any labeling operation execution $L_q^{[k']}$, then $L_p^{[k]} \Rightarrow L_q^{[k']}$.

Intuitive meaning of the above four properties are 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 "$\longrightarrow$". 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 any two Scans ordered by "$\longrightarrow$", 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 [13]. It does imply the

linearizability of the Scans of all processes and labeling operation executions of one process [6]. 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 $\Rightarrow$.

In this paper we are interested in constructing concurrent timestamping systems from Read/Write shared variables (variables, in short). Each such shared variable is written by one process and read by one or more processes. In this paper, 'Write' and 'Read' are used as nouns, referring, respectively, to a write operation execution and a read operation execution, and 'write' and 'read' as verbs. Lamport [18] classifies shared variables in the following three categories.

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 safe 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* or *multivalued* depending upon whether it can hold only boolean or any number of desired values. With these classifications we can define a hierarchy on shared variables, with 1-writer 1-reader boolean safe variable in the lowest level and multiwriter multireader multivalued atomic variable in the highest level. Several researchers [3, 12, 16, 18, 19, 20, 21, 22, 23, 24, 25] have shown how higher level variables can be constructed from the lower level ones.

In the construction of concurrent timestamping systems presented in this paper we use 1-writer multireader atomic, 1-writer 1-reader atomic, 1-writer 1-reader regular, 1-writer multireader safe, and 1-writer 1-reader safe 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 [7]. Each process maintains

a separate local pool of private values. The pools are of infinite size. The private values are totally ordered and are known to all the processes. We would consider values are integer, and the total order is the natural order. The private values of different processes are considered different even if they have the same value.

A label is a vector of n values; its pth component represents a private value of process P_p. The current label of $V[p]$ 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 $V[p]$, process P_p reads all current private values of other processes P_q, namely $l_q[q]$, increments its own private value $l_p[p]$ to obtain a new private value. The new label vector contains these n values, and it is written atomically in $V[p]$. Since the same private value is not used twice in labeling operation executions, no two labels in the system are the same. The ordering of two label vectors is done by using the standard lexicographic order. A Scan simply reads all the current labels and orders them using the lexicographic order. For any 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 unbounded construction satisfies all the properties required for a concurrent timestamping system. (For correctness arguments, please refer to [7].)

In the unbounded construction discussed above, every time a process P_k executes a new labeling operation, it uses a new (so far unused) private value greater than the previously used ones. In a bounded construction, each process has 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 by Dwork and Waarts helps doing the recycling. We quote them verbatim:

> ... for a system to be a concurrent timestamping 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.

We say for any 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 among the private values changes in time. For any 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 $l_p \prec l_q$ iff $l_p[k] \prec_k l_q[k]$. Now, we are envisaged with two things in a bounded construction. First, at any given time there should

not exist in the system two private values of process P_k with the same value that were issued by P_k at different labeling operation executions. Hence, P_k can recycle a private value only if no processes are using it or will be using it shortly. Second, for any two private values v and v' of P_k currently in use, if $v \prec_k v'$ then all other processes should know this ordering. Thus, every time P_k changes the ordering of two different private values, it should inform all the other processes in advance. Roughly speaking, the traceable use abstraction of Dwork and Waarts helps in achieving the above mentioned two objectives. Then, for all labels read by a Scan, the labels are ordered lexicographically, based on the orderings $\prec_i$, for all processes i. The correctness of the bounded system directly follows from that of the unbounded system mentioned above.

Here we present an efficient implementation technique to achieving the above mentioned two objectives using traceable use abstraction. The new construction of concurrent timestamping systems is given in Figure 1.

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. A process P_p executes the LABELING procedure to obtain and assign a new label to $V[p]$, and executes the SCAN function to report the temporal ordering of the labels of $V[1..n]$. A 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 the shared variable. It also uses some private (non shared) variables for each process. We assume that the private variables are persistent.

Let us consider operation executions of a process P_p. In a labeling operation execution, it selects a presently unused private value from its local pool of values, collects the current private values of all other process, and then write these n values in $V[p]$ as its new label. The selection of a new private value is done in such a way that there is no trace of this value in the system. 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

executing the TRACEABLE-WRITE procedure. An execution of the TRACEABLE-READ function (TRACEABLE-WRITE procedure) is called a traceable Read (traceable Write). In a scan operation execution, process P_p first reads the current labels of all the objects, and then determines their temporal ordering using some ordering shared variables. (Incidentally, the structures of the TRACEABLE-WRITE procedure and the TRACEABLE-READ function are quite similar to the WRITE procedure and the READ function, respectively, of the 1-writer multireader shared variable construction of Vidyasankar [25].) The traceable Writes of 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 label values. Immediately after writing a new label value in a main label variable, the process records that variable index in a multireader boolean atomic variable $c[p]$. 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 atomic variables $RC[i,p]$ and $WC[p,i]$. Process P_i sets these values different, by assigning the complement of $WC[p,i]$ to $RC[i,p]$, at the beginning of each traceable Read, and process P_p makes sure that they are the same, at the end of each traceable Write. Hence if the two values are different when the process P_p checks them, then a new traceable Read of P_i must have started. 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 $RC[i,p]$ value to $WC[p,i]$. For each such P_i, P_p takes a note of which of the possible private values of processes P_j could be used by P_i. Finally, it 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 $LEND[p,j]$.

Each traceable Read of process P_p, from a process P_i, after reading $WC[i,p]$ and writing its complement in $RC[p,i]$ as mentioned above, 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 $WC[i,p]$ again and compares with $RC[p,i]$. If the two values continue to be different, it returns the value just read from the main label variable; otherwise, 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, that is, P_i must have done loop iteration p in the first *for*-loop) 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 all the values referred to in $LEND[j,p]$, for all j. 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 $order[p,i]$ which are used by the Scans of P_i.

4 Correctness proof

Proposition 1 [18] *For operation executions B and C on a shared variable, and any operation executions A and D, if $A \longrightarrow B \dashrightarrow C \longrightarrow D$, then $A \longrightarrow D$.*

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. For operation executions A and B executed on the same atomic variable x, we say $A \Longrightarrow_x B$ 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 any operation executions A and D, if $A \longrightarrow B \Longrightarrow_x C \longrightarrow D$, then $A \longrightarrow D$.*

Proof: The relation $B \Longrightarrow_x C$ 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 $O_p^{[k]}(V)$, $k \geq 1$; if it is a Scan (alternatively, a Labeling), we denote it explicitly by $S_p^{[k]}(V)$ (alternatively, $L_p^{[k]}(V)$). The '(V)' 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 $k > 1$, $O_p^{[k-1]} \longrightarrow O_p^{[k]}$. (To avoid ambiguities, we assume the existence of a fictitious operation execution $O_p^{[0]} = L_p^{[0]}$ that writes the initial label. The operation execution $L_p^{[0]}$ took place before non fictitious executions start. The operation executions $L_p^{[0]}$ for all p are concurrent.)

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 $RC[p,i]$ and $WC[p,i]$ are abbreviated to $r_{p,i}$ and $w_{p,i}$, respectively.

Definition. For any 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. We omit the subscript x when it is clear from the context. □

Lemma 1 *No two consecutive labeling operation executions of any process have the same private value. And hence, no two consecutive traceable Writes of any process have the same private value.* □

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

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 $R_i^{[l,1]}(w_{p,i}) \Longrightarrow W_p^{[k]}(w_{p,i}) \Longrightarrow R_i^{[l,2]}(w_{p,i})$ 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 $R_{i,p}^{[l+1]}$ 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 j th traceable Write that writes $w_{p,i}$.

Consider $TW_p^{[k_1]}$. It writes 1 in $w_{p,i}$. This implies 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 $W_i^{[l_1]}(r_{i,p}) \Longrightarrow R_p^{[k_1]}(r_{i,p})$. Note that $TR_{i,p}^{[l_1]}$ reads 0 from $w_{p,i}$ and hence writes 1 in $r_{i,p}$. Also each subsequent traceable Read $TR_{i,p}^{[l'_1]}$, if any, such that $R_i^{[l'_1,1]}(w_{p,i}) \Longrightarrow W_p^{[k_1]}(w_{p,i})$, would read 0 from $w_{p,i}$, and hence will write 1 in $r_{i,p}$. Hence irrespective of whether $W_i^{[l'_1]}(r_{i,p}) \Longrightarrow R_p^{[k_1]}(r_{i,p})$ or $R_p^{[k_1]}(r_{i,p}) \Longrightarrow W_i^{[l'_1]}(r_{i,p})$, on $W_p^{[k_1]}(w_{p,i})$, $w_{p,i} = r_{i,p}$, and if $R_i^{[l,1]}(w_{p,i}) \Longrightarrow W_p^{[k_1]}(w_{p,i}) \Longrightarrow R_i^{[l,2]}(w_{p,i})$ 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 can show in a similar fashion that the assertion holds for $TW_p^{[k_{j+1}]}$. □

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 $R_i^{[l,1]}(w_{p,i}) \Longrightarrow W_p^{[k]}(w_{p,i}) \Longrightarrow R_i^{[l,2]}(w_{p,i})$. 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 $L_p^{[k_{j-1}]} \longrightarrow L_p^{[k_j]}$. They need not be two consecutive operation executions, that is, $k_j \geq k_{j-1} + 1$.

In the following two lemmas, we would show that traceable Reads return valid label values. We also define their reading mapping function π.

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 $R_i^{[l,2]}(w_{p,i}) \Longrightarrow W_p^{[k_{j'}]}(w_{p,i})$, 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 $r_{i,p} \neq w_{p,i}$ and (iii) Lemma 4, it follows that $W_p^{[k_{j''}]}(w_{p,i}) \Longrightarrow R_i^{[l,1]}(w_{p,i})$. That is, $W_p^{[k_{j''}]}(w_{p,i}) \Longrightarrow R_i^{[l,1]}(w_{p,i}) \longrightarrow R_i^{[l,2]}(w_{p,i}) \Longrightarrow W_p^{[k_{j'}]}(w_{p,i})$. 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 $W_i^{[l]}(r_{i,p}) \longrightarrow R_i^{[l]}(c[p]) \Longrightarrow W_p^{[k_{j+1}]}(c[p]) \longrightarrow R_p^{[k_{j+1}]}(r_{i,p})$, we have $W_i^{[l]}(r_{i,p}) \longrightarrow R_p^{[k_{j+1}]}(r_{i,p})$. 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 $W_p^{[k_{j''}]}(w_{p,i}) \Longrightarrow R_i^{[l,1]}(w_{p,i}) \longrightarrow R_i^{[l]}(c[p]) \longrightarrow R_i^{[l,2]}(w_{p,i}) \Longrightarrow W_p^{[k_{j'''}]}(w_{p,i}) \longrightarrow W_p^{[k_j]}(c[p])$. This implies $R_i^{[l]}(c[p]) \longrightarrow W_p^{[k_j]}(c[p])$, 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 $R_i^{[l]}(label[p, x]) \longrightarrow R_i^{[l,2]}(w_{p,i}) \Longrightarrow W_p^{[k_{j'}]}(w_{p,i}) \longrightarrow W_p^{[k_{j+2}]}$. 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, and (iii) $W_p^{[k_j]}(label[p, x']) \longrightarrow W_p^{[k_j]}(c[p]) \Longrightarrow R_i^{[l]}(c[p]) \longrightarrow R_i^{[l]}(label[p, x])$, 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 $R_i^{[l,2]}(w_{p,i})$, and let $TW_p^{[k_j]}$ be the traceable Write such that $R_i^{[l,1]}(w_{p,i}) \Longrightarrow W_p^{[k_j]}(w_{p,i}) \Longrightarrow R_i^{[l,2]}(w_{p,i})$.*

(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 $R_i^{[l,1]}(w_{p,i}) \Longrightarrow W_p^{[k_j]}(w_{p,i}) \Longrightarrow R_i^{[l,2]}(w_{p,i})$. It is clear from the TRACEABLE-WRITE procedure that $TW_p^{[k_j]}$ writes the value in $copylabel[p, i]$ before setting the $w_{p,i}$ and $r_{i,p}$ values equal. The traceable Write $TW_p^{[k_{j+1}]}$ and subsequent traceable Writes of P_p, if they find $r_{i,p} = w_{p,i}$, would not write the copy label variable. From $W_p^{[k_j]}(copylabel[p, i]) \longrightarrow W_p^{[k_j]}(w_{p,i}) \Longrightarrow R_i^{[l,2]}(w_{p,i}) \longrightarrow R_i^{[l]}(copylabel[p, i])$, we have $W_p^{[k_j]}(copylabel[p, i]) \longrightarrow R_i^{[l]}(copylabel[p, i])$. The assertions follow. □

Now we would like to show that private values 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. Thus, all the private values in the existing labels are traceable. 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 at 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.

If the traceable Read $TR_{i,p}^{[l]}$ returns the value v from $copylabel[p,i]$, then, by Lemma 6 and 4, the traceable Write $TW_p^{[k_j]}$ has executed the *if*-statement for process P_i. There it has set $lend[p,p][1][i]$ to v. The successive traceable Writes of P_p that occur before $S_i^{[l]}$ is complete will neither change $lend[p,p][1][i]$ nor reissue v as a new private value.

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 $TW_p^{[k_j]}$, $lend[p,p][1][i]$ is set to v, and in the case of $TW_p^{[k_j+1]}$, $lend[p,p][0][i]$ is set to v. (Note that $TW_p^{[k_j+1]}$ uses a private value different from v.) So, by the argument given in the above paragraph, 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.

It is clear from the TRACEABLE-WRITE procedure that P_q must have got v directly from P_p. Let $L_q^{[m_o]}$ be the corresponding labeling operation execution. 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 $lend[p,p][0..1][q]$. This 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 $lend[p,p][0..1][q]$ 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 $W_q^{[m_o]}(LEND[q,p]) \longrightarrow O_q^{[m_o+1]}(V) \dashrightarrow L_p^{[k_{j'}]}(V) \longrightarrow L_p^{[k_{j'}+1]}$, we have $W_q^{[m_o]}(LEND[q,p]) \longrightarrow L_p^{[k_{j'}+1]}$. That is, $L_p^{[k_{j'}+1]}$ would not reissue v if v is already present in $LEND[q,p]$. Note that $TW_q^{[m_o]}$ will write v in $LEND[q,p][1][q]$ at the end of its execution, and the traceable Write $TW_p^{[k_{j'}]}$ 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 $LEND[q,p][0..1][i]$, and this will not be changed until $S_i^{[l]}$ is complete. Hence $L_p^{[k_{j'}+1]}$ and successive labeling operation executions of P_p that may occur before $S_i^{[l]}$ is complete do not reissue v. □

The following lemma shows that Scans can determine the correct 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 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 timestamping system, we need to make sure that $S_i^{[l]}$ finds $v \prec_p v'$ in case the values are used in ordering some of the scanned labels. From the construction we have $W_p^{[k_{j'}]}(order[p,i]) \longrightarrow TW_p^{[k_{j'}]}(V[p]) \dashrightarrow TR_{i,p}^{[l]}(V[p]) \longrightarrow R_i^{[l]}(order[p,i])$, that is, $W_p^{[k_{j'}]}(order[p,i]) \longrightarrow R_i^{[l]}(order[p,i])$.

Now the question is which private values P_p should store in $order[p,i]$. 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 being used by P_i. In the following we consider a particular value v being used by P_i. Now, there are two cases to be considered. If 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 setting $LEND[p,p][0..1][i]$ to v. Assume P_i obtains v indirectly through another process P_k. From the construction we know that P_k got v directly from P_p. Let the corresponding labeling operation execution be $L_k^{[m_0]}$. Either $TW_p^{[k_j]}$ or $TW_p^{[k_{j+1}]}$ will set $LEND[p,p][0..1][k]$ to v. Now, we need to consider the following three case depending on the overlapping of $L_k^{[m_0]}$ and $S_i^{[l]}$. (1) If $L_k^{[m_0]} \longrightarrow S_i^{[l]}$, then P_k inform P_p that some future Scans ($S_i^{[l]}$ could be one of them) could use v by setting $LEND[k,p][1][k]$ to v. (2) P_k has already informed P_p that P_i could be using v by setting $LEND[k,p][0..1][i]$ to v. (3) P_k has not yet started writing $LEND[k,p]$. But, P_p ($TW_p^{[k_j]}$ or $TW_p^{[k_{j+1}]}$) knows that P_k could be using v through $LEND[p,p][0..1][k]$, and v could also be used by other processes P_i indirectly through P_k.

With the above three observations, we can say that P_p needs to store private values referred to in $LEND[k,p][0..1][i]$, $LEND[k,p][1][k]$ and $LEND[p,p][0..1][k]$ for all k, that is, it needs to reserve at most $5n$ values for p_i. Now, we force the labeling operation executions of P_p, for $v \prec_p v'$, to store v in $order[p,*][x]$ and v' in $order[p,*][y]$ only if $x < y$. Hence on reading $order[p,i]$, P_i will search the $order[p,i][1..5n]$ array starting from index 1 until it finds v and v', and will correctly determine that $v \prec_p v'$. □

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

(a) $W_p^{[k_j]}(c[p]) \Longrightarrow R_{i',p}^{[l']}(c[p])$,

(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 $W_p^{[k_{j-1}]}(w_{p,i'}) \longrightarrow W_p^{[k_j]}(c[p]) \Longrightarrow R_i^{[l]}(c[p]) \longrightarrow R_{i'}^{[l',1]}(w_{p,i'}) \longrightarrow R_{i'}^{[l']}(c[p])$.

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

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

For both the cases we have $W_p^{[k_j]}(c[p]) \Longrightarrow R_{i',p}^{[l']}(c[p])$; part ($a$) follows, and 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 $W_p^{[k_{j-1}]}(w_{p,i'}) \longrightarrow R_{i'}^{[l',1]}(w_{p,i'})$. The part ($b$) follows by Lemmas 4 and 6. □

Theorem 1 *The construction of Figure 1 is a correct implementation of wait-free concurrent timestamping systems.*

Proof: We will 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 $l_q^{[k']}$. 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. If $L_m^{[s_o]} \longrightarrow L_m^{[s_{o'}]}$, then we define $L_p^{[k]} \Rightarrow L_q^{[k']}$.

- Precedence: Without loss of generality we assume $L_p^{[k]} \longrightarrow L_q^{[k']}$. By Lemma 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 $TR_{p,m}^{[k]} \longrightarrow TR_{q,m}^{[k']}$ 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, $L_m^{[s_o]} \longrightarrow L_m^{[s_{o'}]}$. The precedence property follows.

- Consistency: For any two labels $l_p^{[k]}$ and $l_q^{[k']}$ 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 $L_m^{[s_o]} \longrightarrow L_m^{[s_{o'}]}$. The consistency property follows by Lemma 8.

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, $TW_p^{[m_o+1]} \not\longrightarrow TR_{i,p}^{[j]}$, and hence $L_p^{[m_o+1]} \not\longrightarrow S_i^{[j]}$; (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 $R_i^{[j,1]}(w_{p,i}) \Longrightarrow W_p^{[m_o]}(w_{p,i}) \Longrightarrow R_i^{[j,2]}(w_{p,i})$, and so, $TW_p^{[m_o+1]} \not\longrightarrow TR_{i,p}^{[j]}$, and hence $L_p^{[m_o+1]} \not\longrightarrow S_i^{[j]}$.

Monotonicity: Consider two Scans $S_i^{[j]} \longrightarrow S_{i'}^{[j']}$. 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 $S_i^{[j]} \longrightarrow S_{i'}^{[j']}$, we have $TR_{i,p}^{[j]} \longrightarrow TR_{i',p}^{[j']}$. 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 any labeling operation execution $L_q^{[m']}$, if $S_i^{[j]} \longrightarrow L_q^{[m']}$, then $TR_{i,p}^{[j]} \longrightarrow TR_{q,p}^{[m']}$. Then, by Lemma 9($a$), we have $W_p^{[m_o]}(c[p]) \Longrightarrow R_q^{[m']}(c[p])$ and hence, $\pi(TR_{q,p}^{[m']})$ is $TW_p^{[m_o]}$ or its successor. Also by Lemma 5 and 6 and the LABELING procedure, we have $TR_{p,s}^{[m_o]} \longrightarrow TW_p^{[m_o]} \dashrightarrow TR_{i,p}^{[j]} \longrightarrow TR_{q,s}^{[m']}$ for all $s \neq p$, that is, $TR_{p,s}^{[m_o]} \longrightarrow TR_{q,s}^{[m']}$. The extended regularity property follows by Lemma 9(b). □

5 Concluding remarks

We have presented an efficient implementation of the traceable use abstraction of Dwork and Waarts [7], which is oriented towards a new construction of concurrent timestamping systems. Both our and their constructions are similar, but there are a lot of differences. Both the constructions use $O(n \log n)$ size shared variables (*order* and *LEND* variables), where n is the number of processes. Scan and labeling operation executions require $O(n)$ steps. Our construction uses less shared space than that of Dwork and Waarts, and is more efficient. In their construction, 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. Truly speaking, in their construction some labeling operation executions require $O(n^2)$ steps, as the execution of the garbage collection routine requires $O(n^2)$ steps. In our construction, we have used a separate implementation technique for 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 guess

which private values of which processes are in use. Each process needs a separate pool of private values, whose size is fewer than $2n^2$. In their construction, the pool size is $13n^2$. All the ordering shared variables used in our construction are of 1-writer 1-reader type, whereas they are 1-writer n-reader in their construction. In our construction, a Scan reads at most $n-1$ ordering shared variables, whereas in their construction it is $2n-2$.

References

[1] Abrahamson K: On achieving consensus using a shared memory. Proceedings of the Seventh Annual Symposium on Principles of Distributed Computing, 1988, 291–302

[2] Attiya H, Rachman O: Atomic snapshots in $O(n \log n)$ operations. Proceedings of the Twelfth Annual Symposium on Principles of Distributed Computing, 1993, 29–40

[3] Burns JE, Peterson GL: Constructing multi-reader atomic values from non-atomic values. Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing, 1987, 222–231

[4] Chor B, Israeli A, Li M: On processor coordination using asynchronous hardware. Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing, 1987, 86–97

[5] Dijkstra EW: Solution of a problem in concurrent programming control. Communications of the ACM 8:165(1965)

[6] Dolev D, Shavit N: Bounded concurrent time-stamp systems are constructible. Proceedings of the 21st ACM Symposium on Theory of Computing, 1989, 454–466

[7] Dwork C, Waarts O: Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible!. Proceedings of the 24th ACM Symposium on Theory of Computing, 1992, 655–666

[8] Dwork C, Herlihy M, Plotkin S, Waarts O: Time-Lapse snapshots. Proceedings of Israeli Symposium on Theory of Computing and Systems, 1992, LNCS 601, 154–170

[9] Dwork C, Herlihy M, Waarts O: Bounded round numbers. Proceedings of the Twelfth Annual Symposium on Principles of Distributed Computing, 1993, 53–64

[10] Fischer MJ, Lynch NA, Burns JE, Borodin A: Resource allocation with immunity to limited process failure. Proceedings of the 20th IEEE Symposium on Foundations of Computer Science, 1979, 234–254

[11] Gawlick R, Lynch NA, Shavit A: Concurrent timestamping made simple. Israeli Symposium on Theory of Computing and Systems, 1992, LNCS 601, 171-183

[12] Haldar S, Vidyasankar K: A simple construction of 1-writer multireader multivalued atomic variable from regular variables. Tech Rep#9108, Department of Computer Science, Memorial University of Newfoundland, Canada, Aug 1991

[13] Herlihy M, Wing J: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12:463–492(1990)

[14] Israeli A, Li M: Bounded time-stamps. Distributed Computing 6:205–209(1993) (Originally in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, 1987, 371–382)

[15] Israeli A, Pinhasov M: A concurrent time-stamp scheme which is liner in time and space. Proceedings of WDAG, 1992, 95–109

[16] Kirousis LM, Kranakis E, Vitányi PMB: Atomic multireader register. In: van Leeuwen J (ed) Workshop on Distributed Algorithms. Lect Notes Comput Sci, vol 312. Springer, Berlin Heidelberg New York 1987, pp 278–296

[17] Lamport L: A new solution to Dijkstra's concurrent programming problem. Communications of the ACM 17:453–455(1974)

[18] Lamport L: On interprocess communication — Part I: Basic formalism, Part II: Algorithms. Distributed Computing 1:77–101(1986)

[19] Li M, Tromp J, Vitányi PMB: How to share concurrent wait-free variables. CWI Technical Report CS-R8916, (April 1989)

[20] Peterson GL: Concurrent reading while writing. ACM Transactions on Programming Languages and Systems 5:56–65(1983)

[21] Peterson GL, Burns JE: Concurrent reading while writing II: The multiwriter case. Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, 1987, 383–392

[22] Schaffer R: On the correctness of atomic multiwriter registers. Report MIT/LCS/TM-364, 1988, 1–58

[23] Singh, A.K., Anderson, J.H., and Gouda, M.G. The elusive atomic register. To appear in Journal of the ACM. A preliminary version, 'The elusive atomic register revisited', appeared in the Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing, 1987, pp 206–221

[24] Newman-Wolfe R: A protocol for wait-free, atomic, multi-reader shared variables. Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing, 1987, pp 232–248

[25] Vidyasankar K: Concurrent reading while writing revisited. Distributed Computing 4:81–85 (1990)

[26] Vitányi PMB, Awerbuch B: Atomic shared register access by asynchronous hardware. Proceedings of the 27th IEEE Symposium on Foundations of Computer Science, 1986, 233–243

Declarations

Constants:

n = number of processes;

Type:

label-type: array [1..n] of integer;

Shared variables:

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

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

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

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

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

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

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

$cl[p]$: 0..1; {**Initially** 0}
lend: array $[p, 1..n]$ of array [0..1] of label-type; {**all initially 0**}

old-label: label-type; {**all Initially** 0, except $old\text{-}label[p] = 1$}

Figure 1: Shared variables.

```
Procedure TRACEABLE-WRITE(p: 1..n; new-label: label-type );
var
    i, j: 1..n; {loop index}
    lr: boolean;
begin
    cl[p] := ¬cl[p];
    write new-label in label[p, cl[p]];
    write cl[p] in c[p];
    for i := 1 to n do
                                              {could be done in parallel}
        begin
          read lr from RC[i, p];
          if lr ≠ WC[p, i] then
            write new-label in copylabel[p, i];
            for j := 1 to n do lend[p, j][0..1][i] := ⟨old-label[j], new-label[j]⟩;
            write lr in WC[p, i];                  {WC[p, i] = RC[i, p]}
          endif;
        endfor;
    for j := 1 to n do lend[p, j][1][p] := new-label[j];
    for j := 1 to n do write lend[p, j] in LEND[p, j]
    old-label := new-label;
end; {of procudure}

Function TRACEABLE-READ(p: 1..n, i: 1..n): label-type;
var
    lw: 0..1;
    lc: 0..1;
    savelabel: label-type;
begin
    read lw from WC[i, p];
    write ¬lw in RC[p, i];                    {RC[p, i] ≠ WC[i, p]}
    read lc from c[i];
    read savelabel from label[i, lc];
    read lw from WC[i, p];
    if (RC[p, i] ≠ lw) then return(savelabel)
                                              {RC[p, i] = WC[i, p]}
    else
      read and return(copylabel[i, p])
    endif;
end; {of function}
```

Figure 1: Construction for process p. (Cont'd.)

```
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;
    private-value: integer;
begin
    for j := 1 to n do
        read temp[j] from LEND[j, p]; {we do not need temp[j][0][j]}
    select a new private-value not in temp and the current private value;
    for j := 1 to n do
        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
        and write them in order[p, j];
    for j := 1 to n, j ≠ p, do lab[j] :=TRACEABLE-READ(p, j);
    TRACEABLE-WRITE(lab[1][1], lab[2][2], ..., lab[p][p] := private-value, ..., lab[n][n]);
end;

Procedure SCAN(p: 1..n):(l̄, ≺);
var
    i, j, k: 1..n;
    lab: array [1..n] of label-type;
begin
    for j := 1 to n do lab[j] :=TRACEABLE-READ(p, j);
    for i := 1 to n do
        for j := 1 to n do
            let k be the least significant index in which lab[i] differs from lab[j];
            if order[k, p] is not read yet then read it;
            determine the order between lab[i] and lab[j];
end;
```

Figure 1: Construction for process p. (Cont'd.)