Bounded Concurrent Time-Stamp Systems Are Constructible

Danny Dolev* Nir Shavit†

Abstract

Concurrent time stamping is at the heart of solutions to some of the most fundamental problems in distributed computing. Based on concurrent-time-stamp-systems, elegant and simple solutions to core problems such as *fcfs*-mutual-exclusion, construction of a multi-reader-multi-writer atomic register, probabilistic consensus,... were developed. Unfortunately, the only known implementation of a concurrent time stamp system has been theoretically unsatisfying, since it requires unbounded size time-stamps, in other words, unbounded memory. Not knowing if bounded concurrent-time-stamp-systems are at all constructible, researchers were led to constructing complicated problem-specific solutions to replace the simple unbounded ones. In this work, for the first time, a bounded implementation of a concurrent-time-stamp-system is presented. It provides a modular unbounded-to-bounded transformation of the simple unbounded solutions to problems such as above. It allows solutions to two formerly open problems, the bounded-probabilistic-consensus problem of Abrahamson [A88] and the *fifo-ℓ*-exclusion problem of [FLBB85], and a more efficient construction of *mrmw* atomic registers.

*IBM Almaden Research Center and Hebrew University Jerusalem.

†Hebrew University, Jerusalem. Supported by a Libnitz Foundation Scholarship and Israeli Communications Ministry Award. Currently visiting the *TDS* group at *MIT*, supported by *NSF* contract no CCR-8611442, by *ONR* contract no N0014-85-K-0168, by *DARPA* contract no N00014-83-K-0125, and a special grant from *IBM*. Parts of this research were also conducted while the author was visiting *AT&T* Bell Laboratories and *IBM* Almaden Research Center.

Keywords: Concurrency, Time Stamping, Atomic Registers, Serialization.

1 Introduction

The paradigm of concurrent time stamping is at the heart of solutions to some of the most fundamental problems in coordination of concurrent processes [A88, CIL87, D65, DGS88, H88, L74, PB87, VA86].

A time stamp system of n asynchronous processes is traditionally conceived as consisting of n label registers, one per process, written by it and read by all others. The labels are unbounded natural-numbers, where each process can execute infinitely many labeling and scan operations on the label registers. A *labeling* operation is a sequence of reads of other labels, followed by a write of a label greater than the maximal value read. The label values written, establish a total order on all labeling operations ever executed. A *scan* operation is a sequence of reads of all process' labels, returning a subset of labels ordered consistently with this total ordering. A *concurrent-time-stamp-system* (*ctss*) is a time-stamp-system in which any number of labeling or scan operations (by different processes) may overlap in time. A major requirement is that labeling and scan operations of any process be waitfree, that is, completed in finite time independently of the pace of other processes.

Concurrent time stamping is the basis for simple solutions to a wide variety of the basic problems in concurrency control. Examples of such problems include *fcfs*-mutual-exclusion, construction of a multi-reader-multi-writer atomic

register, probabilistic consensus,... Unfortunately, the only known implementation of the above paradigm is based on labels of unbounded size. This is a major drawback, since bounded memory size is a key requirement of the problems at hand, implying these elegant and simple unbounded solutions have little theoretical value. Since it was unknown whether bounded concurrent-time-stamp-systems are constructible, researchers were led to devising complicated problem-specific solutions to show that the above problems are solvable in a bounded way [Bl87, BP87, CIL87, D65, DGS88, FLBB79, FLBB85, K78, L74, L86d, LH88, LV88, R86, P81, P83, PB87, VA86].

Israeli and Li in [IL87] were the first to isolate the notion of bounded-time-stamping as an independent concept, developing an elegant theory of bounded *sequential*-time-stamp-systems, that is, time-stamp systems in a world where no two operations are ever concurrent. They also devised a concurrent labeling scheme in which the labels provide a causality preserving relation. However, this relation is not a total ordering since unrelated labels and cycles are possible. Moreover, this scheme deals only with labeling, and does not address the central problem of how labels can be scanned concurrently, therefore lacking some of the key properties of concurrent-time-stamp-systems.

In this paper, for the first time, a bounded construction of a concurrent-time-stamp-system is presented. It allows a modular transformation of the simple unbounded solutions to such core problems as above[1]. It provides a powerful tool, enabling the design of simple unbounded concurrent-time-stamp based algorithms, with the knowledge that such unbounded solutions immediately imply the bounded ones[2]. This is exemplified by providing the basis to solutions of the above flavor [ADMS88, ADS89] to two formerly open problems, the bounded-probabilistic-consensus problem of [A88] (requiring to solve the probablistic-consensus problem of [CIL87] without using an atomic coin-flip operation), and the *fifo-ℓ*-exclusion problem of [FLBB79]. The only known solutions to the latter problem [DGS88, P88], achieve weaker forms of fairness than the original *test and set* based solution of [FLBB79].

Though one might think that the price of introducing such a powerful modular transformation would be a blowup in memory size or number of operations, this is hardly the case. The construction presented in the paper requires n registers of $O(n)$ bits each, meeting the lower bound of [IL87] for sequential-time-stamp-system construction. Though because of lack of space, a complete comparison table cannot be provided in this paper, one example of the efficiency of the *ctss* solutions is given by the famous problem of multi-reader-multi-writer atomic register construction. A simple solution based on transforming the unbounded [VA86] protocol (See *Appendix A* for a description), has the same space complexity of the only proven algorithm [PB87, S88], yet a better time complexity, $O(n)$ memory accesses for a write, $O(n \log n)$ for a read, as compared with $O(n^2)$ for either in the former. Concurrent time stamp systems are informally defined in *Section 2*, and implemented in *Section 3*. Rigorous formal definitions and correctness proofs based on the formalism of Lamport [L86a, L86c] will be presented in the full paper.

2 Concurrent Time Stamping

To provide the reader with a better intuition for the more abstract formal definitions presented later, the properties of a concurrent-time-stamp-system are first outlined informally via the example of its unbounded *natural-number* based implementation.

Informally, the natural-number based *ctss* consists of n registers of unbounded size, each written by one of n asynchronous processes and read by all others. The labels are natural numbers with the usual ordering among them[3]. Each process can execute infinitely many *labeling* or *scan* operations, any number of them concurrently with the operations of other processes. The scan

[1] See *Appendix A*.

[2] Bounded time-stamp algorithms for a message passing environment without faults are very similar to that described in this paper. Lack of space prevents us from describing it.

[3] Process id's are added lexicographically to break symmetry, a well known technique which will be referred to in the sequel.

is the operation of collecting a set of labels ℓ, one of each process, by executing a sequence of reads of the labels in an arbitrary order. The labeling operation is simply a collecting of all the labels followed by a write of $max(\ell)+1$. The labels written during labeling operations are monotonically increasing, and, though some were possibly created concurrently with others, define a total order on all labeling operations ever performed. Since for any two labeling operations that are non-concurrent, the order among the labels reflects the order among the operations, this order defines the manner in which all labeling operations could be serialized. Though no process ever knows all of this order, the order among the subset of labels returned by any scan is in fact the same as the total ordering on all the labeling operations[4], no matter how many labeling operations occurred while the labels were being scanned!

A *Concurrent Time Stamp System* is an abstract data type shared among n concurrent and completely asynchronous processes. There are two *waitfree* (see [H88, AG88]) operations that any process can execute on the *ctss*, a *labeling* operation and a *scan* operation. Assume that each process' program consists of these two operations, whose execution generates a sequence of *elementary operation executions*, totally ordered by the *precedes* relation (of [L86a, L86c], denoted " $\longrightarrow$ "), and were any number of scan operation executions are allowed between any two labeling operation executions. The following

$$L_i^{[1]} \longrightarrow S_i^{[1]} \longrightarrow L_i^{[2]} \longrightarrow L_i^{[3]} \longrightarrow S_i^{[2]} \longrightarrow S_i^{[3]} \longrightarrow S_i^{[4]} \longrightarrow \ldots$$

is an example of such a sequence by process i, where $L_i^{[k]}$ denotes process i's k^{th} execution of a labeling operation, and $S_i^{[k]}$ the k^{th} execution of a scan operation (the superscript $[k]$ is used for notation, and is not visible to the processes). A *global time model*[5] of operation executions is assumed.

With each labeling operation execution $L_i^{[k]}$, a *label* $\ell_i^{[k]}$ is associated. A *scan* operation returns a pair $(\bar{\ell}, \prec)$, where the *label view* $\bar{\ell} = \{\ell_1^{[k_1]} \ldots, \ell_n^{[k_n]}\}$ is an ordered set of labels[6] (one per process), and $\prec$ is an *irreflexive total order* among them, such that:

P1 *ordering:* There exists an *irreflexive total order* $\Longrightarrow$ on the set of all labeling operations, such that:

- **a.** *precedence:* For any pair of labeling operation executions $L_p^{[a]}$ and $L_q^{[b]}$ (where possibly $p = q$), if $L_p^{[a]} \longrightarrow L_q^{[b]}$, then $L_p^{[a]} \Longrightarrow L_q^{[b]}$.
- **b.** *consistancy:* For any scan operation execution $S_i^{[k]}$ returning $(\bar{\ell}, \prec)$, $\ell_p^{[a]} \prec \ell_q^{[b]}$ if and only if $L_p^{[a]} \Longrightarrow L_q^{[b]}$.

The above property formalizes the idea that a *ctss* can be envisioned as a black box, inside which hides a mechanism (a logical clock) associating causally ordered time stamps – from an infinite totally ordered range – with each of the labeling operations, and where scanning is like peeping into this black box, each scan returning a view of a part of this hidden ordering. The black box metaphor is used to stress that it suffices to know of the existence of such a total ordering $\Longrightarrow$, while the ordering itself need not be known.

One should bear in mind that the asynchronous nature of the operations allows situations where a scan overlaps many consecutive labeling operations of other processes. Also, several consecutive scans could possibly be overlapped by a single labeling operation. It is therefore important that a requirement be made that the label view $\bar{\ell}$ returned by $S_i^{[k]}$ be a meaningful one, namely, reflecting the ordering among labeling events immediately before or concurrent with the scan, and not just any possible set of labels. This will

[4] This property is simple to achieve using unbounded labels, since the ordering among the labeling operations is just the ordering among the labels. The fact that such a property is achievable using bounded size labels is somewhat baffling, since as the example in *Section 3* shows, the order among the labeling operations cannot be the order among the labels.

[5] Implying that for any two operations, $a \longrightarrow b$ or $b \longrightarrow a$ (for more details see [L86c, B88]).

[6] For the purposes of many of the applications (such as atomic register construction), one should allow the label to include an associated value field, denoted $value@\ell_i^{[k]}$. For the sake of simplicity, discussion of how this added feature is implemented will be differed to the appendix.

eliminate uninteresting trivial solutions and introduce a measure of liveness into the system. This requirement is formalized in the following definition, where $\dashrightarrow$ is the *can affect* relation of [L86a,L86c].

P2 *regularity*: For any label $\ell_p^{[a]}$ in $\bar{\ell}$ of $S_i^{[k]}$, $L_p^{[a]} \dashrightarrow S_i^{[k]}$, and there is no $L_p^{[b]}$ such that $L_p^{[a]} \longrightarrow L_p^{[b]} \longrightarrow S_i^{[k]}$.

Though such a *regular* concurrent time stamp system (*P1-P2*) would suffice for some applications (as in Lamport's "Bakery Algorithm" [L74]), a more powerful *monotonic* concurrent time stamp system will be needed in applications such as the *Multi-Reader-Multi-Writer Atomic Register* construction (as in [VA86]). To this end the following third property is added:

P3 *monotonicity*: For any label $\ell_p^{[a]}$ in $\bar{\ell}$ of $S_i^{[k]}$, there does not exist an $S_j^{[k']}$ with a label $\ell_p^{[b]}$ in its label view $\bar{\ell}'$, such that $S_i^{[k]} \longrightarrow S_j^{[k']}$ and $L_p^{[b]} \longrightarrow L_p^{[a]}$ (possibly $i = j$).

It is important to note that *P3* does not imply that labeling and scan operations of all processes are serializable. It does however imply the serializability of the scans of all processes and labeling of any *one* process. The scans "behave" as if the labels of any process are monotonically increasing, in the sense that a scan returns a label of a labeling operation that is at least as late as that of any labeling operation of a label returned in the scans preceding it. In the following section, a bounded implementation of a concurrent time stamp system from atomic registers is presented and informally justified. Rigorous definitions[7] and correctness proofs will appear in the full version.

3 The Implementation

The description of the implementation is divided into two parts, the implementation of the labeling operation, and the implementation of the scan.

[7] The above definitions do not include, for example, initialization conditions of the system.

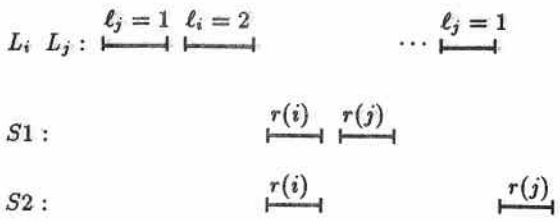

Figure 1: Scan Concurrent with Sequential Labelings

The key property of the labeling operation is to allow establishing the causality-preserving total order $\Longrightarrow$ among all labeling operation executions. Though it is not required that a process "knows" what this order is, it is required that the set of labels that it "chooses" during a system execution is such, that an almighty outside observer, given a description of the execution and based on the labels, would be able to reconstruct $\Longrightarrow$. This almighty observer could thus view all labeling operation execution intervals as if they were shrunk to points, that is, as if they were completely sequential.

Requiring this property alone, will however not be sufficient. As *Example 3.1* shows, even if all labeling operations are sequential, since labels are from a bounded range (and therefore the same labels are reused), a process scanning the labels concurrently with ongoing labeling operations, cannot deduce the order $\Longrightarrow$ from the order of the labels alone.

Example 3.1. In *Figure 1*, segments represent operation execution intervals, where time runs from left to right. Two processes i and j perform labeling operations sequentially, j followed by i, followed by many labelings, till eventually the labels are reused, and j for example uses the same label as before. A third process z performs a scan concurrently with the labelings, reading ℓ_i and then ℓ_j. $S1$ and $S2$ represent possible executions of this same scan, the only difference being that many labeling operations of other processes occurred between the reads in $S2$. In both the

case that the scan is of the form $S1$ and the case that it is of the form $S2$, the values collected are $\ell_i = 2$ and $\ell_j = 1$, where the order among the labels is, say $1 < 2$. However, in the case of $S1$, j's labeling preceded i's, while in $S2$, i's labeling preceded j's. Thus, the order of the labels is not the order among the labeling operations, introducing an unresolvable ambiguity.

Faced with the above ambiguity, it is clear that in order to design a scan operation, the properties of labeling operation implementation should be such, that even though the order $\Longrightarrow$ between any pair $L_x^{[a]}$ and $L_y^{[b]}$ is not conveyed by the order of their associated labels, the labels do provide enough information to allow an implementation of a scan operation. The new implementation will not require that by reading a pair of labels of processes i and j, one will be able to establish the order among their associated labeling operation executions. Instead, it will be required that by reading the labels of i and j more than once (yet only a constant number of times), one will be able to choose from all the labels read, a label of i and a label of j, for which the order $\Longrightarrow$ among the labeling operation executions can in fact be deduced. In the following sections, after presenting these additional properties, a scan operation implementation that utilizes them will be shown.

The basic communication primitive used in the presented implementations is a *single-writer-multi-reader atomic register*. Constructions of such registers from weaker primitives have been shown in [L86a, L86b, BP87, IL87, N87]. The *concurrent-time-stamp-system* will consist of n *swmr* atomic registers v_i, $i \in \{1..n\}$, each v_i written by process i, read by all, and having values in some range V. In the unbounded natural number implementation of a *ctss*, V is just the unbounded set of natural numbers, and $\stackrel{\nu}{\prec}$ for any labeling is the usual irreflexive total ordering among them. In the following subsections, the set of possible label values V, together with an irreflexive and antisymmetric relation $\stackrel{\nu}{\prec}$ among them, are defined in terms of a *precedence graph*[8] $(V, \stackrel{\nu}{\prec})$. Each possible label value is a node in this graph. The order among the labels in any two registers is the order $\stackrel{\nu}{\prec}$ established by the edges of the precedence graph. Based on the precedence graph, an implementation of the labeling and scan operations will then be provided. Unlike in the unbounded natural number implementation, and following the above discussion, the returned ordering $\prec$ among labeling operations is not the same as the ordering $\stackrel{\nu}{\prec}$.

[8] see [IL87] for lower bounds on the size of such graphs.

3.1 The Labels and the Precedence Relation

The following is the description of the *precedence graph* T^n. Though the precedence graph (of unbounded size) defined by the natural numbers is acyclic, this will not be true for T^n.

Define A *dominates* B in G, where A and B are two subgraphs of a graph G (possibly single nodes), to mean that all nodes of A have edges directed to all nodes of B. Define the following generalization of the composition operator of [IL87]. The *α-composition*, $G \circ_\alpha H$, of two graphs G and H, where α is a subset of the nodes of G, is the following non-commutative operation:

> Replace every node $v \in \alpha$ of G by a copy of H (denoted H_v) and let H_v (or v) dominate H_u in $G \circ_\alpha H$ if v dominates u in G.

Define the graph T^2 to be the following graph of 5 nodes: a cycle of three nodes {3,4,5} (where 3 dominates 5, which dominates 4, which in turn dominates 3), all dominating the nodes {2,1}, where node 2 in turn, dominates node 1.

Define the graph T^k (a complete tournament) inductively to be:

1. T^1 is a single node.
2. $T^k = T^2 \circ_\alpha T^{k-1}$, where $\alpha = \{5, 4, 3, 1\}$ and $k > 1$.

The graph $T^n = (V, \stackrel{\nu}{\prec})$ is the precedence graph to be used in the implementation of the labeling and scan algorithms of a concurrent time-stamp system for n processes. For any process i, each node in T^n corresponds to a uniquely defined label value ℓ_i. The label can be viewed as a string $\ell_i[n..1]$ of n digits, where each $\ell_i[k] \in \{1 \ldots 5\}$ is the digit of the corresponding node in

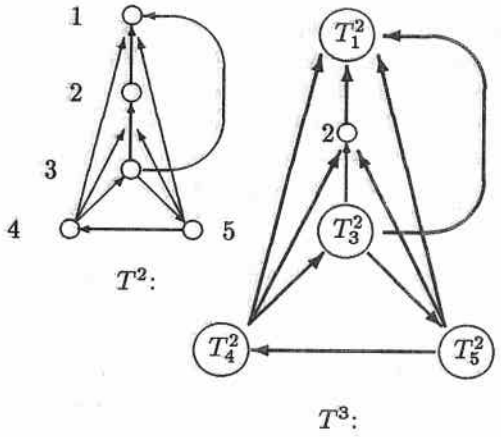

Figure 2: The Recursive Graph Structure for T^2 and T^3

T^2, replaced by a T^k subgraph during the k^{th} step of the inductive construction above. The digit $\ell_i[n]$ is always 1, representing the complete T^n graph, and if in ℓ_i, $\ell_i[k] = 2$, then $\ell_i[j] = 1$ for all $j \in \{k-1..1\}$ (since node 2 is never expanded in the induction step). Therefore, given any label ℓ_i, the T^k subgraph of T^n in which its corresponding node is located is identified by the corresponding prefix $\ell_i[n..k]$.

To assure that based on the graph T^n a total ordering among the label values returned by a scan can be established, one needs to break symmetry among processes having the same label. As usual, *process-ids* are used. Thus, the label ℓ_i is assumed to be concatenated with the *id* of process i. The label and *id* are lexicographically ordered. This, in terms of the graph T^n, amounts to no more than assuming that each T^1 graph consists of a total order tournament of n nodes, each process i always choosing the ith node in the order. For the sake of simplicity this point is not elaborated on in the sequel.

3.2 The Labeling Operation

Let the *collect* operation by any process i be a reading of all the registers v_j, $j \in \{1..n\}$, once each, in an arbitrary order returning a label set ℓ (not to be confused with $\bar{\ell}$, the output label view of a scan operation). The *labeling* operation of a process i is of the form described below, where

$\mathcal{L} : V^n \times \{1..n\} \mapsto V$ is a *labeling function*, returning a label value ℓ_i "greater than" all other label values[9]. This is a form similar to the *natural number ctss*, where the labeling function $\mathcal{L}$ is just $max(\ell) + 1$. However, the interpretation of "greater than" is not as straightforward as in the *natural number* case.

```
procedure labeling;
  begin
    ℓ := collect;
    v_i := L(ℓ, i)
  end;
```

The definition of the labeling function $\mathcal{L}(\ell, i)$ presented below, is based on a recursively defined function $\mathcal{L}^k(G, \ell, \ell_x)$, which, given a T^k subgraph G, of T^n, a set of labels ℓ, and a "maximal" label $\ell_x \in \ell$ in T^k, returns the label of a node in G that is, as termed above, "greater than" the other labels. For the sake of simplicity, and since the collected set of labels ℓ remains unchanged in $\mathcal{L}(\ell, i)$ once it is collected (similarly the variable ℓ_x, once it is computed), it is treated as a global variable and is not passed as a parameter in all the utility functions used by $\mathcal{L}(\ell, i)$. The following functions are used in defining $\mathcal{L}$:

$num\_labels(G)$ – a function that, for the given label set ℓ, returns how many of the labels are in sub-graph G;

$dom(x)$ – a function that, for a given digit $x \in \{1..5\}$ representing a node in the graph T^2, returns the next dominating node; namely, $dom(1) = 2$, $dom(2) = 3$, $dom(3) = 4$, $dom(4) = 5$ and $dom(5) = 3$;

$dominating\_set(\hat{\ell}, \ell_i)$ – a function that, for a set of labels $\hat{\ell} \subseteq \ell$, and a label $\ell_i \in \hat{\ell}$, returns a subset of labels $\{\ell_j \in \hat{\ell} \mid \ell_i \not\succ \ell_j\} \cup \{\ell_i\}$; and

$max(\hat{\ell})$ – a function that, for a set of labels $\hat{\ell} \subseteq \ell$, returns a label

$$(\ell_x \in \hat{\ell} : |dominating\_set(\hat{\ell}, \ell_x)| \leq |dominating\_set(\hat{\ell}, \ell_j)|, \forall \ell_j \in \hat{\ell}),$$

the maximal label, i.e., the one least dominated within this set.

[9] Initially, all labels are on node 111..11, the node dominated by all others in T^n.

Denote the *concatenation operation*, where G is a string and x is a digit, by $G.x$. The following is thus the definition of the labeling function $\mathcal{L}(\ell, i)$. The subgraphs G are identified with the relative prefixes, where T^n is identified with the label 1:

function $\mathcal{L}(\ell, i)$;
 function $\mathcal{L}^k(G)$;
 begin
 1: **if** k= 1 **then return** G;
 2: **if** $\ell_x[n..k] \neq G$
 then return $\mathcal{L}^{k-1}(G.1)$;
 3: **if** $\ell_x[n..k-1] = G.2$
 then return $\mathcal{L}^{k-1}(G.3)$;
 4: **if** $k > 2$ **then**
 if $\ell_x[k-2] \in \{2,3,4,5\}$ **and**
 $(\ell_i[n..k-1] \neq \ell_x[n..k-1])$
 then return $\mathcal{L}^{k-1}(G.dom(\ell_x[k-1]))$;
 5: **if** $(num\_labels(\ell_x[n..k-1]) < k-1)$ **or**
 $((num\_labels(\ell_x[n..k-1]) = k-1)$ **and**
 $(\ell_i[n..k-1] = \ell_x[n..k-1]))$
 then return $\mathcal{L}^{k-1}(G.\ell_x[k-1])$
 else return $\mathcal{L}^{k-1}(G.dom(\ell_x[k-1]))$;
 end $\mathcal{L}^k$;
begin
 $\ell_x := max(dominating\_set(\ell, \ell_i))$;
 return $\mathcal{L}^n(T^n)$;
end $\mathcal{L}$;

For the purpose of giving the reader some intuition about the properties of the labeling operation, let it be assumed that one can talk about the values of the labels of all processes at "points in time". Though the goal in the remainder of this section is to show how the labeling operation executions allow to define the order $\Longrightarrow$, it will first be shown that they meet a much simpler requirement. The requirement is that at any point in time, the subgraph of the precedence graph T^n induced by the labeled nodes (those whose corresponding label is written in some v_i), contains no cycle. Since T^n is a complete tournament, this implies that at any point in time, all labels are totally ordered.

The labeling operation executions maintain two "invariants," namely, that at any point in time (1) there are labels on at most two of the three nodes in any cycle of any subgraph T^k (the cycle consists of "supernodes" {3,4,5}, called supernodes since they are actually T^{k-1} subgraphs), and (2) there are no more than k labels in the cycle of any subgraph T^k. Maintaining the second invariant is the key to maintaining the first, and the first implies that at any point in time, there are never any cycles among labels.

The manner by which the invariance of (1) and (2) is preserved, is explained via several examples. In these examples, T^3 is a precedence graph for a system of three processes x, y and z. All examples start at a point in time where $\ell_x^{[a]} = 134$, $\ell_y^{[b]} = 135$, and $\ell_z^{[c]} = 141$, that is, all labels are totally ordered by $\prec$.

Example 3.2. Assume that the following sequence of labeling operation executions occur sequentially. Process y performs $L_y^{[b+1]}$, reading $\ell_x^{[a]}$, $\ell_y^{[b]}$ and $\ell_z^{[c]}$, and moving, based on $\mathcal{L}(\ell, y)$ to $\ell_y^{[b+1]} = 142$. Process z performs $L_z^{[c+1]}$, reading the new label $\ell_y^{[b+1]}$, and thus moving to the T^2 subgraph 14, ($L_y^{[b+2]} = 144$, $L_z^{[c+2]} = 145$, $L_y^{[b+3]} = 143$...), maintaining the above invariants, because the T^2 graph is a precedence graph for 2 processes. If at some point x moves, in $L_x^{[a+1]}$ it will read the labels of both z and y as being in the T^2 subgraph 14. Since $num\_labels('14')$=2, by line 5 of $\mathcal{L}(\ell, i)$, x will move to $\ell_x^{[a+1]} = 151$.

The reader can convince himself that following any labeling operation execution $L_z^{[c]}$ by some process z, the above invariants hold, and that for the set ℓ of labels that were read in $L_z^{[c]}$'s collect operation (denoted $read(L_z^{[c]})$), it is the case that $(\forall \ell_y^{[b]} \in read(L_z^{[c]}))(\ell_y^{[b]} \prec \ell_z^{[c]})$, that is, the new label chosen is greater than all those read.

As seen in the following example, in the concurrent case, more than k labels may move into the same T^k structure at the same time. It is thus not immediately clear why the second invariant holds.

Example 3.3. Assume that the following sequence of labeling operation executions occur concurrently. Processes x and y begin performing $L_x^{[a+1]}$ and $L_y^{[b+1]}$ concurrently, reading $\ell_x^{[a]}$, $\ell_y^{[b]}$ and $\ell_z^{[c]}$ and computing $\mathcal{L}$, such that $\ell_x^{[a+1]} = \ell_y^{[b+1]} = 142$. If they then continue to complete their operations by writing their labels, though they have the same node as a label, they were

concurrent, and can be ordered by relative *id*. If any of them then continued to perform a new labeling operation, since *num_labels*('14') > 2, it would choose label 151, not entering the cycle. However, let us suppose that they do not both complete writing their labels, that is, x stops just before writing $\ell_x^{[a+1]}$ to v_x, while y writes $\ell_y^{[b+1]} = 142$. Process z then performs $L_z^{[c+1]}$, reading the new label $\ell_y^{[b+1]}$ and the old label $\ell_x^{[a]}$, thus moving to $L_z^{[c+1]} = 143$. Processes y and z continue to move into and in the cycle of the T^2 subgraph 14, since they continue to read x's old label. Then, at some point x completes $L_x^{[a+1]}$, and there are three labels in 14 (two of them in the cycle). However, if x now performs a new labeling $L_x^{[a+2]}$, it will read the labels of both x and y as being in 14. Since *num_labels*('14') > 2, by line 5 of $\mathcal{L}(\ell, i)$, x will move to $\ell_x^{[a+2]} = 151$, not entering the cycle.

Generalizing the above example, even if many processes move into a T^k subgraph, without reading one another's labels, at most k of them will enter the cycle in T^k. The reason is the following well known *flag principal*[10]:

> If $k+1$ people, each first raise a flag, and then count the number of raised flags, at least one person must see $k+1$ flags raised.

By the definition of the labeling function $\mathcal{L}$, each process moving into the cycle of a T^k subgraph, must first move to either supernode 1 or 2 in T^k, only then can it perform a labeling into the cycle. The move to 1 or 2 is the raising of the flag, and the move into the cycle is the counting of all flags.

The following example shows that even though by the above, there are at most k labels at a time in any T^k structure, the sets of labels *read* in a labeling operation execution, may contain cycles.

Example 3.4. Process z begins performing $L_z^{[c+1]}$, reading $\ell_x^{[a]} = 134$. Process y then performs $L_y^{[b+1]}$, reading $\ell_x^{[a]}$, $\ell_y^{[b]}$ and $\ell_z^{[c]}$, and moving to $\ell_y^{[b+1]} = 142$. Process x performs $L_x^{[a+1]}$, reading the new label $\ell_y^{[b+1]}$ and $\ell_z^{[c]}$, and thus by line 5 of $\mathcal{L}$, moving to $\ell_x^{[a+1]} = 151$. Process y then performs $L_y^{[b+2]}$, reading $\ell_x^{[a+1]}$ and moving to $\ell_y^{[b+2]} = 152$. Finally, process z reads $\ell_y^{[b+2]}$. It thus read $\ell_x^{[a]} = 134$, $\ell_y^{[b+2]} = 152$, and $\ell_z^{[c]} = 141$, three labels on a cycle.

In order to select a label dominating all others, z must establish where the "maximal label" among them is. To overcome the problem that the labels read form cycles (as in the above example), the labeling function $\mathcal{L}(\ell, z)$ does not take into account "old values" such as $\ell_x^{[a]}$; it considers only the labels that dominate the current label $\ell_z^{[c]}$. In order to maintain the first invariant, z should move to $\ell_z^{[c+1]} = 131$, to dominate the current labels of both x and y. However, there is seemingly a problem, since z did not read the label $\ell_x^{[a+1]} = 151$, and so, how can it decide what label to choose in order to dominate $\ell_x^{[a+1]} = 151$? The solution is due to the fact that z can deduce the existence of $\ell_x^{[a+1]} = 151$, since in all of the cycle of T^3 there are 3 labels, and in order to move to $\ell_y^{[b+1]} = 152$, y must have read some label in node 151 of the T^2 subgraph 15. By simple elimination this must be the label of x. This simple rule is maintained by application of line 4 in $\mathcal{L}$. However, if the above scenario occurred in the cycle of a T^k graph, where $k > 3$, then in order to allow the same reasoning as above, it must be that if z read $\ell_y^{[b+2]} = 152$ (or $\ell_y^{[b+2]} \in \{153, 154, 155\}$), it can conclude that $k-2$ other labels were read by $L_y^{[b+2]}$ in the T^{k-1} subgraph 15. It is for this purpose that supernode 1 of any T^k graph where $k > 2$, is *not* a single node, but a T^{k-1} subgraph. A process can thus choose the node 2, only after it established that there were $k-1$ labels in supernode 1. Since node 2 is a "bridge," that some process must "cross" (choose) before any process can move into the cycle, the above reasoning holds.

Though the above invariants hold, it follows from *Example 3.4* that the property that the chosen new label is greater than all those read, true for sequential labeling operation executions, does not hold in the concurrent case. Fortunately, there is a similar property that does hold, a property that will prove important in the implementation of the scan. Let the notation $r_j(L_i^{[k]})$ and

[10] Proof follows by the fact that the last person to start counting flags must have seen $k+1$ flags raised.

$w(L_i^{[k]})$ denote the read of v_j and write of v_i during a labeling operation execution $L_i^{[k]}$ by a process i.

Definition 3.1. *Labeling $L_x^{[a]}$ is observed by $L_y^{[b]}$ (denoted $L_x^{[a]} \xrightarrow{obs} L_y^{[b]}$) if $r_x(L_y^{[b]}) = \ell_x^{[a]}$ or there exists an $L_z^{[c]}$ such that $r_z(L_y^{[b]}) = \ell_z^{[c]}$ and $L_x^{[a]} \xrightarrow{obs} L_z^{[c]}$.*

The relation $\xrightarrow{obs}$ is actually the transitive closure of the *read* relation. Let $maximal\_obs(L_x^{[a]})$ be the set of operation executions

$$\{L_y^{[b]} \mid y \in \{1..n\},\ L_y^{[b]} \xrightarrow{obs} L_x^{[a]} \text{ and } (\forall L_y^{[b']})(\text{if } L_y^{[b]} \longrightarrow L_y^{[b']} \text{ then } L_y^{[b']} \not\xrightarrow{obs} L_x^{[a]})\},$$

that is, including the "latest" label observed for each process. In the concurrent executions, instead of the new label being greater than all the labels read, it is the case that

$$(\forall \ell_y^{[b]} \in maximal\_obs(L_x^{[a]}))(\ell_y^{[b]} \overset{v}{\prec} \ell_x^{[a]}),$$

namely, the new label chosen is greater than the latest of those observed. For the labeling $L_z^{[o+1]}$ of *Example 3.4*, though z read $\ell_x^{[a]} = 143$, and $\ell_z^{[o+1]} \overset{v}{\prec} \ell_x^{[a]}$, it is the case that its maximal observed label is $\ell_x^{[a+1]}$, and $\ell_x^{[a+1]} \overset{v}{\prec} \ell_z^{[o+1]}$.

Finally, the following is the irreflexive total order $\Longrightarrow$ on the labeling operation executions as required by property *P1*.

Definition 3.2. *Given any two distinct labeling operation executions $L_x^{[a]}$ and $L_y^{[b]}$, $L_x^{[a]} \Longrightarrow L_y^{[b]}$ if either*

1. $L_x^{[a]} \xrightarrow{obs} L_y^{[b]}$, *or*
2. $L_x^{[a]} \not\xrightarrow{obs} L_y^{[b]}$, $L_y^{[b]} \not\xrightarrow{obs} L_x^{[a]}$, *and* $\ell_x^{[a]} \overset{v}{\prec} \ell_y^{[b]}$.

Intuitively, since with every $L_x^{[a]}$ there is an associated label $\ell_x^{[a]}$, $\Longrightarrow$ is a "lexicographical" order on a pairs $(L_x^{[a]}, \ell_x^{[a]})$. The first element in the pair is ordered by $\xrightarrow{obs}$, a partial order that is consistant with the ordering $\longrightarrow$ (if $L_x^{[a]} \longrightarrow L_y^{[b]}$ then in $L_y^{[b]}$, y read $\ell_x^{a]}$ or a later label). The second element is ordered by $\overset{v}{\prec}$, an irreflexive and antisymmetric relation. In the full paper it is proven, that the "static" relation $\overset{v}{\prec}$ on the labels, completes the "dynamic" partial order $\xrightarrow{obs}$ to a total order on all labeling operation executions.

3.3 The Scan Operation

The scan algorithm consists of two main steps, performing a sequence of $8n \log n$ *collect* operations [11], and analyzing the collected labels to select a set $\bar{\ell}$ for which an order $\prec$ can be returned.

Let $\ell^{c,m,k}, c \in \{1..8\}$, $m \in \{1..\lceil \log n \rceil\}$, and $k \in \{1..n\}$ denote variables, each holding a set of labels $\{\ell_1^{c,m,k}, .., \ell_n^{c,m,k}\}$ collected in the c^{th} *collect* operation execution of the m^{th} *level* of the k^{th} *phase*. Let $half(r)$ and $other\_half(r)$ be complementary functions, that for a given set r, return two disjoint subsets $r1$ and $r2$, such that $r1 \cup r2 = r$ and $-1 \leq |r1| - |r2| \leq 1$.

The scan algorithm returns the set of labels $\bar{\ell}$, one of each process, and the ordering $\prec$ among them is represented by the vector O holding a *permutation* of numbers in $\{1..n\}$, the number in the i^{th} position representing the relative order of the label ℓ_i [12].

```
function scan;
    function select(m,k,r);
    begin
        if |r| = 1 then return (x : x ∈ r);
          else
            x := select(m−1, k,half(r));
            y := select(m−1, k, other_half(r));
            if (∃c1, c2 ∈ {1..8})
              (c1 < c2) ∧ (ℓ_x^{c1,m,k} ≺^v ℓ_y^{c2,m,k})
              then return y
              else return x
            fi;
        fi;
    end select;
begin
    R := {1..n};
    O[1..n] := 0;
    ℓ̄ := ∅;
    for k := 1 to n do
```

[11] Note that the *scan* algorithm requires a scanning process *only to read* other labels, and does not require it to write. This lack of a need for two way communication between the scanner the labelers is a property found in the implementation of the *natural number* based *ctss*.

[12] For the sake of simplicity, though the returned labels in $\bar{\ell}$ could contain various data associated with the given labeling operation (that is, data written into the register v_i together with the implementation label value), the scan implementation, will return only the implementation label value ℓ_i.

```
        for m := 1 to ⌈log n⌉ do
            for c := 1 to 8 do
                ℓ^{c,m,k} := collect
            od;
        od;
    od;
    for k := n downto 1 do
        s := select(⌈log n⌉, k, R);
        ℓ̄ := ℓ̄ ∪ {ℓ_s^{8,⌈log n⌉,k}};
        O[s] := k;
        R := R − {s};
    od;
    return (ℓ̄, O);
end scan;
```

The scan operation, as noted above, begins with a sequence of $8n\lceil \log n \rceil$ collect operations, for which the returned labels are all saved in a set of variables $\ell^{c,m,k}, c \in \{1..8\}$, $m \in \{1..\lceil \log n \rceil\}$, and $k \in \{1..n\}$. The remainder of the algorithm defines how to choose n of these labels, one per process, for which $\prec$ (i.e. $\Longrightarrow$) can be established. The following is an outline of how this selection process is performed.

By the order of label collection, the labels read in phase $k = 1$ are the earliest to have been collected, those for $k = n$ the last. From the $8\lceil \log n \rceil$ collected label sets of each phase, the algorithm selects one label. The selected label in the k^{th} phase will be the k largest in the order $\prec$. As it turns out, to guarantee that this is the case, it suffices that the following *Condition 1* holds (slightly abusing notation in the definition):

> For the label $\ell_s^{8,\lceil \log n \rceil,k}$, collected in the $\lceil \log n \rceil^{th}$ level of the k^{th} phase, and any label $\ell_y^{8,1,k}$ of a process $y \in R$, collected in the 1^{st} level of the k^{th} phase, it is the case that $L_y^{8,1,k} \Longrightarrow L_s^{8,\lceil \log n \rceil,k}$.

Maintaining *Condition 1* is sufficient to assure that the label returned in the k^{th} phase is the k largest. Let it be shown that the labeling operation execution of a label returned in a phase $k' < k$, preceded (in the ordering $\Longrightarrow$) that of the label returned in the phase k. The following shows that this is the case for the labels $\ell_x^{8,\lceil \log n \rceil,k}$, $\ell_y^{8,\lceil \log n \rceil,k-1}$ and $\ell_z^{8,\lceil \log n \rceil,k-2}$ returned in phases k, $k-1$, and $k-2$ respectively. The same line of proof can be extended inductively to all $k' < k$.

By *Condition 1*, $L_y^{8,1,k} \Longrightarrow L_x^{8,\lceil \log n \rceil,k}$. Since the read of $\ell_y^{8,1,k}$ was performed after that of $\ell_y^{8,\lceil \log n \rceil,k-1}$, either the label of the same labeling operation execution was read in both cases, or $L_y^{8,\lceil \log n \rceil,k-1} \Longrightarrow L_x^{8,\lceil \log n \rceil,k}$. By similar reasoning $L_z^{8,\lceil \log n \rceil,k-2} \Longrightarrow L_y^{8,\lceil \log n \rceil,k-1}$, which by transitivity of $\Longrightarrow$, establishes $L_z^{8,\lceil \log n \rceil,k-2} \Longrightarrow L_x^{8,\lceil \log n \rceil,k}$.

The *select* function applied in any phase, is a recursively defined "winner take all" type selection algorithm, among all the processes in R. It returns the *id* of the "winner," a process s meeting *Condition 1*. At any level m of the application of select $select(m, k, r)$, the winners of the selections at level $m-1$ are paired up, and from each pair one "winner" process is selected, to be passed on to the $(m+1)^{\text{th}}$ level of selection. After at most $\lceil \log |R| \rceil$ levels, s, the winner of all selections, is returned.

Based on the definition of the *select* function, maintaining the following *Condition 2* suffices to assure that the label of the process s returned by $select(m, k, r)$, meets *Condition 1*.

> Of the two processes x and y in the application of *select* at level m of phase k, the one returned, say x, is such that $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$, where $\ell_y^{1,m,k}$ and $\ell_x^{8,m,k}$ respectively are the labels associated with these labeling operation executions.

Maintaining *Condition 2* suffices for the following reason. If at level m process x was selected between x and y, and at level $m-1$ process y was selected between y and z, by the same line of proof as above, from $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$ and $L_z^{1,m-1,k} \Longrightarrow L_y^{8,m-1,k}$, it follows that $L_z^{8,m-2,k} \Longrightarrow L_x^{8,m,k}$. By induction this implys *Condition 1*.

Recall *Example 3.1*, implying that it is impossible to establish the order $\Longrightarrow$ among two labeling operation executions, from the order among their associated labels alone. To overcome this problem, instead of attempting to decide the order between two given labeling operation executions, the algorithm will choose a pair out of

several given labeling operation executions, for which the order $\Longrightarrow$ can be determined. Thus, to allow the *select* operation at level m of phase k, to choose a "winner" process, say x, for which $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$, labels of x and y from 8 consecutive collects will be analyzed.

Let it first be shown that if the following *Condition 3* holds for y, that is

$$(\exists c1, c2 \in \{1..8\})(c1 < c2) \wedge (\ell_x^{c1,m,k} \overset{v}{\prec} \ell_y^{c2,m,k}),$$

then $L_x^{c1,m,k} \Longrightarrow L_y^{c2,m,k}$ (this, because of the order of label collecting, will imply $L_x^{1,m,k} \Longrightarrow L_y^{8,m,k}$). Assume by way of contradiction that $L_x^{c1,m,k} \Longrightarrow L_y^{c2,m,k}$. Since $\ell_x^{c1,m,k} \overset{v}{\prec} \ell_y^{c2,m,k}$, it must be by the definition of $\Longrightarrow$ that $L_y^{c2,m,k} \overset{obs}{\longrightarrow} L_x^{c1,m,k}$. It cannot be that $\ell_y^{c2,m,k} \in maximal\_obs(L_x^{c1,m,k})$, since by the properties of the labeling scheme, for the label $\ell_y^{[b]} \in maximal\_obs(L_x^{c1,m,k})$, $\ell_y^{[b]} \overset{v}{\prec} \ell_x^{c1,m,k}$. Thus, there must be a different labeling operation execution $\ell_y^{[b]} \in maximal\_obs(L_x^{c1,m,k})$, $L_y^{c2,m,k} \longrightarrow L_y^{[b]}$. This label $\ell_y^{[b]}$ was already observed (i.e. must have been written), before the end of the read of $\ell_x^{c1,m,k}$. Thus, $\ell_y^{[b]}$, or a label later than it, must have been read instead of $\ell_y^{c2,m,k}$, in the *collect* $c2$ of level m in phase k, a contradiction.

It remains to be shown that if *Condition 3* does not hold for y, it is the case that $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$, and x can be correctly returned. Assume by way of contradiction that *Condition 3* does not hold for y. It cannot, by the same arguments as above, be that *Condition 3* holds for x, that is, $(\exists c1, c2 \in \{1..8\})(c1 < c2) \wedge (\ell_y^{c1,m,k} \overset{v}{\prec} \ell_x^{c2,m,k})$. Therefore, it must be that there are four nonconsecutive collects of $\ell^{c1,m,k}$, $c1 \in \{1,3,5,7\}$, and four nonconsecutive collects of $\ell^{c2,m,k}$, $c2 \in \{2,4,6,8\}$ such that the labels $\ell_y^{c1,m,k}$, $c1 \in \{1,3,5,7\}$ are all different from one another, and the labels $\ell_x^{c2,m,k}$, $c2 \in \{2,4,6,8\}$ are all different from one another. The reason is that if any two of them are the same, say $\ell_y^{3,m,k}$ and $\ell_y^{5,m,k}$, then in order for the above *Condition 3* not to hold for x $c1 = 4$ and $c2 = 3$, it must be that $\ell_x^{4,m,k} \overset{v}{\prec} \ell_y^{3,m,k}$. But since $\ell_y^{3,m,k}$ and $\ell_y^{5,m,k}$ are the same, it would follow that $\ell_x^{4,m,k} \overset{v}{\prec} \ell_y^{5,m,k}$, and *Condition 3* would hold for y, a contradiction.

To complete the proof, it remains to be shown that if the labels $\ell_y^{c1,m,k}$, $c1 \in \{1,3,5,7\}$ are all different from one another, and the labels $\ell_x^{c2,m,k}$, $c2 \in \{2,4,6,8\}$ are all different from one another, then $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$. The situation above is such that during the 8 collect operations, each of the processes x and y executed a new labeling operation at least 3 times. It can be formally shown[13] that the third new labeling operation execution $L_x^{8,m,k}$, after x and y moved at least 3 times, occurred completely after the initial labeling of y, that is, $L_y^{1,m,k} \longrightarrow L_x^{8,m,k}$.

Formal proofs will be presented in the full paper. As a final comment, note that for algorithms where only the maximum label is required, and not a complete order among all returned labels (like in construction of a *mrmw* atomic register or solutions to the *mutual exclusion* problem), only one phase of label collection is required, that is, only $8 \log n$ collects[14].

4 Acknowledgements

We would like to thank Yehuda Afek and Mike Merritt for many important conversations and comments. It was a subtle observation of Mike's regarding pairwise-consistancy among scans, that led us eventually to the current *ctss* definitions.

References

[ADMS88] Y, Afek, D. Dolev, M. Merritt, and N. Shavit, "A Bounded fifo solution to the l-exclusion problem," in preparation.

[A88] K. Abrahamson, "On Achieving Consensus Using a Shared Memory," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 291-302.

[AG88] J. H. Anderson, and M. G. Gouda, "The Virtue of Patience: Concurrent Programming With and Without Waiting," unpublished manuscript, Dept. of Computer Science, Austin, Texas, Jan. 1988.

[ADS89] H. Attiya, D. Dolev, and N. Shavit, "A Bounded Probabilistic Shared-Memory Consensus Algorithm," unpublished manuscript.

[13] This claim is *not* true if less than 3 new labelings took place.

[14] The number of collects in each phase can be lowered to $5 \log n$, if one gives up the property that the order of reads in a collect be arbitrary).

[B88] S. Ben-David, "The Global Time Assumption and Semantics for Concurrent Systems," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 223-231.

[Bl87] B. Bloom, "Constructing two-writer atomic registers," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 249-259.

[BP87] J. E. Burns, and G. L. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 222-231.

[CIL87] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware", *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 86-97.

[D65] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *CACM 8, 9*, 1965, p. 569.

[DGS88] D. Dolev, E. Gafni, and N. Shavit, "Toward a Non-Atomic Era: L-Exclusion as a Test Case," *Proc. 20th Annual ACM Symp. on the Theory of Computing*, 1988.

[FLBB79] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Resource Allocation with Immunity to Limited Process Failure," *Proc. 20th IEEE Symp. on Foundations of Computer Science*, 1979, pp. 234-254.

[FLBB85] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Distributed fifo Allocation of Identical Resources Using Small Shared Space," *MIT/LCS/TM-290*, 1985.

[H88] M. P. Herlihy, "WaitFree Implementations of Concurrent Objects," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 276-290.

[IL87] A. Israeli and M. Li, "Bounded Time Stamps," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science*, 1987, pp. 371-382.

[K78] H. P. Katseff, "A New Solution to the Critical Section Problem," *Proc. 10th Annual ACM Symposium on the Theory of Computing*, 1978, pp. 86-88.

[L74] L. Lamport, "A new Solution of Dijkstra's Concurrent programming problem," *CACM 17, 8* 1974, pp. 453-455.

[L86a] L. Lamport, "On Interprocess Communication. Part I: Basic Formalism," *Distributed Computing 1, 2* 1986, 77-85.

[L86b] L. Lamport, "On Interprocess Communication. Part II: Algorithms," *Distributed Computing 1, 2* 1986, pp. 86-101.

[L86c] L. Lamport, "The Mutual Exclusion Problem. Part I: A Theory of Interprocess Communication," *J. ACM 33, 2* 1986, pp. 313-326.

[L86d] L. Lamport, "The Mutual Exclusion Problem. Part II: Statement and Solutions," *J. ACM 33, 2* 1986, pp. 327-348.

[LV88] M. Li, and P. Vitanyi, "Uniform Construction for Wait-Free Variables," unpublished manuscript, 1988.

[LH88] E. A. Lycklama and V. Hadzilacos, "A Fair Mutual Exclusion Algorithm With Small Communication Variables," submitted for publication, 1988.

[N87] R. Newman-Wolfe, "A Protocol for Wait-free Atomic, Multi Reader Shared Variables," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 232-248.

[P81] G. L. Peterson, "Myths about the Mutual-Exclusion Problem," *IPL 12, 3* 1981, pp. 115-116.

[P83] G. L. Peterson, "Concurrent Reading While Writing," *ACM TOPLAS 5, 1* 1983, pp. 46-55.

[PB87] G. L. Peterson, and J. E. Burns, "Concurrent Reading While Writing II : The Multi-Writer Case," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science*, 1987, pp. 383-392.

[P88] G. L. Peterson, personal communication.

[R86] M. Raynal, *Algorithms for Mutual Exclusion*, North Oxford Academic, 1986.

[S88] R. Schaffer, "On the Correctness of Atomic Multi-Writer Registers," MIT/LCS/TM-364, June 1988.

[SAG87] A. K. Singh, J. H. Anderson and M. G. Gouda, "The Elusive Atomic Register Revisited," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp 206-221.

[VA86] P. Vitanyi, and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science*, 1986, pp. 233-243.

A Some Examples of Applications

The following is a simple unbounded algorithm for solving the famous problem of constructing a *mrmw atomic register* , from *swmr atomic registers*. This solution is a version (due to Li and Vitani [LV88]) of the elegant and simple unbounded Vitani-Awerbuch algorithm [VA86]. It is based on the use of a *natural number* ctss. Each process i writes to a *mrsw* atomic register denoted v_i. Each register contains two fields, a *label*, that is, a *natural number*, and a *value* associated with it (*value@label* using the notation of [LV88]). The following is an implementation of the *read* and *write* by a process i.

```
function read;
begin
    read v_1, .., v_n;
    select the maximal time-stamp ℓ_x;
    return value@ℓ_x;
end;

procedure write(value);
begin
    read v_1, .., v_n;
    select the maximal time-stamp ℓ_x;
    write into v_i the value and ℓ_x + 1;
end;
```

Note that the *write* operation is just a *labeling*, and the *read* is a *scan* followed by returning the value associated with the maximal label. As mentioned earlier, one would need to let the labels of the *ctss* include their associated values. Replacing the above unbounded operations by the Labeling and Scan operations of the bounded concurrent-time-time-stamp system will immediately produce a bounded solution to the problem. Note again that the general implementation of the scan operation, as described in this *extended abstract* requires $8n \log n$ collects, but since only the maximum (and not a total ordering) of the labels is required, it can be reduced to $8 \log n$ collects, as will be elaborated upon in the full paper.

The following is a *fifo* solution to the *l-Exclusion Problem* due to [ADMS88], based on the use of a *ctss*. In the following, the *scan* and *label* operations of process i are as described, where the *ctss* is implemented using *swmr atomic registers*, and x_i, $i \in \{1, .., n\}$ are *swmr safe* registers.

```
do forever
    x_i := true;
    labeling;
L: (ℓ̄, ≺) := scan;
    if |{ j | x_j ∧ (ℓ_j ≺ ℓ_i)}| ≥ l then goto L fi;
    critical section
    x_i := false;
    remainder section
od;
```

The only known bounded *fifo* solution to the problem, due to [FLBB79] was based on the use of a strong form of *Test and Set*. It was unknown whether a level of fairness higher than n^2-*waiting* (see [DGS88]) without use of *test and set* can be achieved. It is interesting to note that the amount of shared memory needed meets the lower bound of [FLBB79]. If one is interested in the unbounded implementation, just substitute $\ell_i := max(\ell_1, .., \ell_n) + 1$ for the *labeling* operation, and $read(\ell_1, .., \ell_n)$ for the *scan*. Notice that for $l = 1$, the above is a very simple solution to the fundamental *mutual exclusion problem* of [D65]. Other algorithms such as the unbounded implementation of a *ctss* in the *Bakery Algorithm* of Lamport [L74], can also be modularly replaced, and by adding a simple modification to allow the *ctss* to include restarts, the solution can be made to be resiliant to *restart failures* [L74, L86d].