Lecturer: Nir Shavit

Handout 6

1 A Concurrent Time Stamp System

The following is a formal definition of a *ctss* for a system of processes numbered $1 \ldots n$. It uses the axiomatic specification formalism of Lamport [L86a,L86c]. The reader may benefit by checking how the formal properties described below are met by the unbounded implementation described in the next section.

A *ctss* is a problem specification with an operational interface. A *ctss* that permits n concurrent operations has $2n$ operation types, specifically $labeling_i(\ell_i)$ and $scan_i(\bar{\ell}, \prec)$ for $i \in \{1 \ldots n\}$. A $labeling_i$ operation associates an input *value*, ℓ_i, taken from any domain, $\mathcal{D}$, with a label.[1] We call ℓ_i the *labeled-value* of operation $labeling_i$. In an application such as an atomic register construction, the labeled-value would be the value written to the register, while in a mutual exclusion type application, where the input values are unimportant, it would be null. A $scan_i$ operation returns as output a pair $(\bar{\ell}, \prec)$ where the *view* $\bar{\ell} = \{\ell_1 \ldots, \ell_n\}$ is an indexed set of labeled-values (one per process), and $\prec$ is an *total order* on these indexes.

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). The labeled-value input in each labeling operation execution $L_i^{[k]}$ is denoted as $\ell_i^{[k]}$. [2] A *global time model* of operation executions is assumed, implying that for any two operation executions $a \longrightarrow b$ or $b \dashrightarrow a$ (for more details see Section **??**).

The elementary operation executions of a *ctss* must have following set of properties:

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

a. *precedence:* For any pair of labeling operation executions $L_p^{[a]}$ and $L_q^{[b]}$ (where possibly p and q are the same process), if $L_p^{[a]} \longrightarrow L_q^{[b]}$, then $L_p^{[a]} \Longrightarrow L_q^{[b]}$.

[1] In order correctly handle initial conditions the value domain $\mathcal{D}$ must specify some initial value.

[2] In order for a unique labeled-value $\ell_i^{[k]}$ to be associated with each label operation execution $L_i^{[k]}$, the reader can think of $\ell_i^{[k]}$ as a triplet $< \ell_i^{[k]}, i, k >$, where the second and third fields are dummy indexes, used only for purposes of the specification.

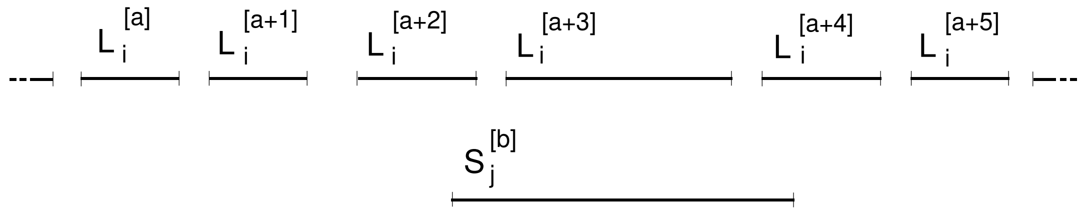

Figure 1: Regularity.

b. *consistency:* For any scan operation execution $S_i^{[k]}$ returning $(\bar{\ell}, \prec)$, $p \prec q$ 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 of which hides a mechanism (a logical clock) associating causally ordered time stamps – from an infinite totally ordered range – with each of the labeled-values entered in labeling operations, and where scanning is like peeping into this black box, each scan returning a view of a part of this hidden ordering[3]. 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 operation execution overlaps many consecutive labeling operation executions of other processes. Also, several consecutive scans could possibly be overlapped by a single labeling operation execution. It is therefore important that a requirement be made that the view $\bar{\ell}$ returned by $S_i^{[k]}$ be a meaningful one, namely, reflecting the ordering among labeling operation executions immediately before or concurrent with the scan, and not just any possible set of labeled-values (In the example of Figure 1, any of the labeled-values $\ell_x^{[a+1]}$ to $\ell_x^{[a+4]}$ can be returned by $S_i^{[k]}$, but not those preceding or following them). This will 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 labeled-value $\ell_p^{[a]}$ 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 [LV88,VA86]). To this end the following third property is added:

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

[3]Notice that there is no requirement that labeled-values returned by different scans be comparable.

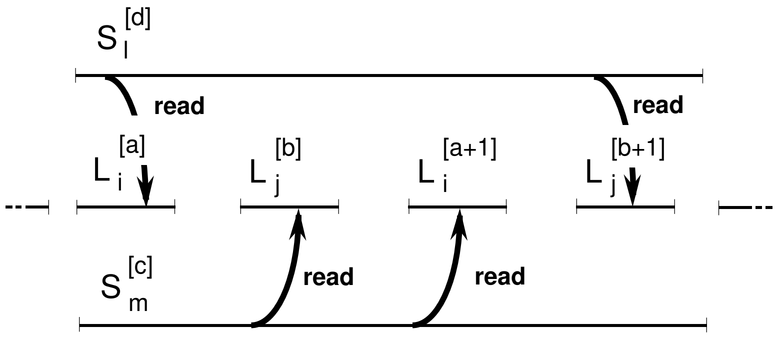

Figure 2: Monotonicity does not imply Atomicity.

Monotonicity is the property that in the unbounded natural number ctss can be described by saying that the labels of any one process, as read by increasingly later scans, are "monotonically non-decreasing." In other words, later scans cannot read labels smaller than those read by earlier ones. It is important to note though, that *P3* does not imply that labeling and scan operation executions of all processes are serializable, that is, appear to happen atomically (Figure 2 shows two scan operations meeting property *P3*, that can not be serialized). It does however imply the serializability of the scan operation executions of all processes relative to the labeling operation executions of any *one* process. Property *P4* is an extension of part of the regularity property to the $\Longrightarrow$ order [4]. The properties *P3* and *P4* together imply that all *scan* operations that consider only the "largest" value, where "largest" is based on the $\prec$ ordering, can be serialized with respect to all labeling operations.

P4 *order regularity*: For any labeled-value $\ell_p^{[a]}$ in $\bar{\ell}$ of $S_i^{[k]}$, $S_i^{[k]} \longrightarrow L_q^{[b]}$ implies that $L_p^{[a]} \Longrightarrow L_q^{[b]}$.

2 Unbounded Concurrent Time Stamping

The basic communication primitive used in our implementations is a *single-writer-multi-reader atomic register* (*swmr* atomic register). Our goal is to design an implementation that is *wait-free* [H88,AGa88]: each process' *scan* or *label* operation executions consists of a bounded number of *swmr* register operations, independently of the pace or type of operations carried out by other processes. Wait-free constructions of *swmr* atomic registers from weaker primitives have been shown in [BP87,IL87,L86b,SAG87,N87].

We begin with the following a simple implementation of a *ctss* using *swmr* registers of unbounded size. The concurrent-time-stamp-system will consist of n *swmr* atomic registers v_i, $i \in \{1..n\}$. Each v_i is written by process i and read by all. Each *labeling*$_i$ operation writes ℓ_i to register v_i. In our implementation, ℓ_i is a data-type consisting of two fields, a labeled-value, denoted *value*(ℓ_i), and its associated label, denoted *label*(ℓ_i). Each *label*(ℓ_i) is

[4]The need for property *P4* in applications such as the *multi-reader-multi-writer* atomic register construction of [LV88,VA86], was discovered by Gawlick [Rainer-thesis].

procedure *labeling*(*val*);
  begin
    ℓ := *collect*;
    $v_i := (val,\ (\max_{j\in\{1..n\}} number_j + 1, i))$;
  end;
function *scan*;
  begin
    ℓ := *collect*;
    return $(\{value(\ell_1)\ldots value(\ell_n)\}, ord(\ell))$;
  end;

Figure 3: The unbounded *natural number* based implementation.

a pair of the form $(number_i, i)$, where $number_i$ is a natural number, and $i \in \{1..n\}$ is the id of the process writing ℓ_i.

A process i collects the labels and values of other processes by performing a *collect* operation, a reading of all the registers v_j, $j \in \{1..n\}$, once each, in some arbitrary order. The collect operation returns an indexed set $\ell = \{\ell_1, .., \ell_n\}$, that is, one value and associated label per process. The collected elements in ℓ are ordered by $ord(\ell)$, an ordering on their indexes in $\{1..n\}$, such that i is smaller than j if and only if the label $(number_i, i)$ is lexicographically smaller than the label $(number_j, j)$. Figure 3 provides the pseudo-code of the *labeling* and *scan* operations for a process i.

To understand how property $P1$ is met, consider that if the labeling operation execution of ℓ_i by a process i completely preceded the labeling operation execution of ℓ_j by j, then it must be that j chose a label with $number_j > number_i$, since j collected ℓ_i. If they are concurrent, at worst they might both collect the same maximal label, and choose $number_i = number_j$, in which case they are ordered by their *id*s. Thus, the lexicographic order on the labels defines a linearization order [HW88] on the concurrent labeling operation executions, that is, an order $\Longrightarrow$ by which they can be thought of as happening sequentially in time. The reader can convince herself that properties *P2*-*P4* follow directly from the use of *swmr* atomic registers in the implementation.

It is important to note that the actual $label(\ell_1)\ldots label(\ell_1)$ used in computing $ord(\ell)$ are hidden from the user (scan operations do not return them), and there is thus no way to compare the order among a pair of values returned by different scans.

3 A Bounded Concurrent Time-Stamp System

3.1 Labels and Precedence

The bounded implementation presented will be of the exact same form as the unbounded natural number based one. 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 and read by all. Each value ℓ_i written to register v_i, consists, just as in the unbounded case, of two fields, a *labeled-value*, to which the input of a labeling operation is written, and an associated *label*.

Note In the sequel, almost all of the discussion involves only the label field of v_i and not its labeled-value field. In order to simplify the exposition, we choose, with few exceptions, to ignore the existence of the labeled-value field, and deal only with the associated label field. Thus for example, the notation $\ell_i^{[k]}$ will represent only the label field written in a labeling operation execution $L_i^{[k]}$. We trust that the interested reader will be able to add the relevant operations regarding the labeled-value, as in the unbounded implementation in Section 2.

Let V denote the range of possible labels, and $\overset{V}{\prec}$ an irreflexive and antisymmetric relation among them. In the unbounded natural number implementation of a *ctss*, V is just the unbounded size set of pairs of natural number and integer in $\{1..n\}$, and $\overset{V}{\prec}$ is the lexicographic total ordering among them. In the following sections, the set of possible label values V of the implementation, together with a relation $\overset{V}{\prec}$ among them, are defined in terms of a *precedence graph*[5] $(V, \overset{V}{\prec})$. Each possible label is a node in this graph. The order among the labels in any two registers is the order $\overset{V}{\prec}$ established by the edges of the precedence graph. A tournament is a complete directed graph. The precedence graph representing labels of the natural number based implementation, is an acyclic tournament of unbounded size, i.e., a total order. The definition of the precedence graph will provide the basis for describing the implementation of the labeling and scan operations.

3.2 A Bounded Precedence Graph

The following is the description of the *precedence graph* T^n (see Figure 4). Unlike the unbounded precedence graph defined by the natural numbers, T^n contains cycles.
Define A *dominates* B in G, where A and B are two subgraphs of a graph G (possibly single nodes), to mean that every node of A has edges directed to every node 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}, and where node 2 in turn, dominates node 1.
Define the graph T^k (a 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 $T^n = (V, \overset{V}{\prec})$ 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

[5] The elegant idea of defining the labels and ordering as a tournament graph, was introduced by Israeli and Li in [IL87].

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 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. Thus, the label ℓ_i is assumed to be concatenated with the *id* of process i, where 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 i^{th} node in the order. For simplicity this point is not further elaborated upon in the sequel).

3.3 The Labeling Operation

Recall that the *collect* operation by any process i is a reading of all the registers v_j, $j \in \{1..n\}$, once each, in an arbitrary order, returning a set ℓ of labels. The *labeling* operation of a process i is of the form described in Figure 5, 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[6]. This is the same form as the natural number *ctss*, where the labeling function $\mathcal{L}$ returns $(\max_{j\in\{1..n\}} number_j + 1, i)$. However, the interpretation of being "greater than" is not as straightforward as in the natural number case.

The definition of the labeling function $\mathcal{L}(\ell, i)$ presented below, is based on a recursively defined function $\mathcal{L}^k(G, \ell, \ell_{max})$, which, given a T^k subgraph G, of T^n, a set of labels ℓ, and a "maximal" label $\ell_{max} \in \ell$ in T^k, returns the label of a node in G that is "greater than" the other labels. They reader may benefit by going through Examples 7, 3.2, 3.3 before or during the reading of the code in Figure 6. For simplicity, and since the collected set of labels ℓ remains unchanged in $\mathcal{L}(\ell, i)$ once it is collected (similarly the variable ℓ_{max}, 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 $\{\ell_j \in \hat{\ell} \mid \ell_i \preceq\!\!\!\!/\ \ell_j\} \cup \{\ell_i\}$; and

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

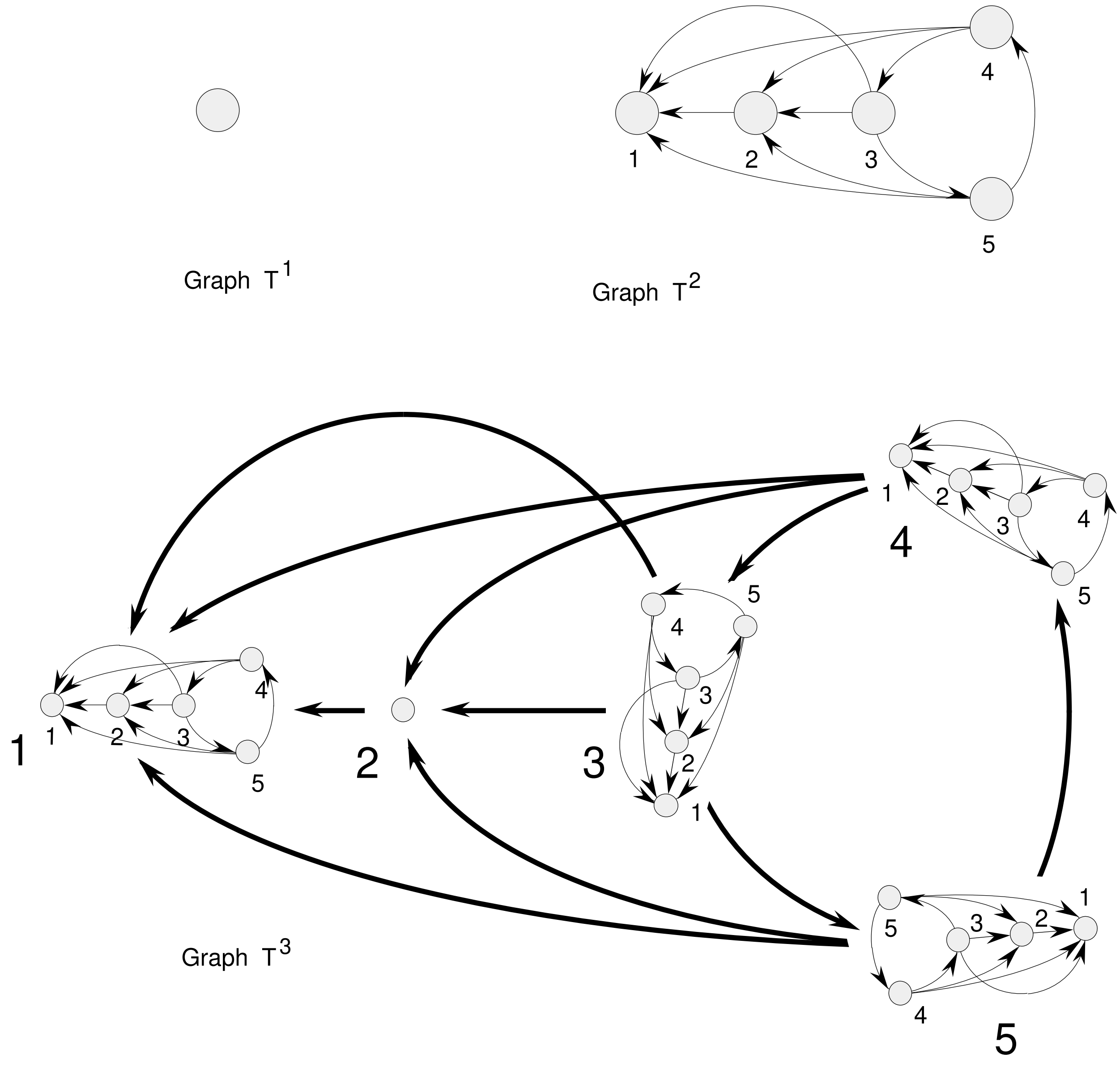

Figure 4: The precedence graph.

$max(\hat{\ell})$ – 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.

procedure *labeling*(*val*);
  begin
    ℓ := *collect*;
    v_i := (*val*, $\mathcal{L}(\ell, i)$);
  end;

Figure 5: The labeling operation.

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

Figure 6: The labeling function.

Define $G.x$ to be the *concatenation* of string G and digit x. Figure 6 is thus the definition of the labeling function $\mathcal{L}(\ell, i)$, where the parameter subgraphs G are identified with the relative label prefixes, and T^n is identified with the label 1. To give 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". To show how the labeling operation executions allow to define the order $\Longrightarrow$, we will first argue informally that they meet a much simpler requirement, namely, that at any point in time:

(r1) The labels reflect the precedence among non-concurrent labeling operation executions.

(r2) 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.

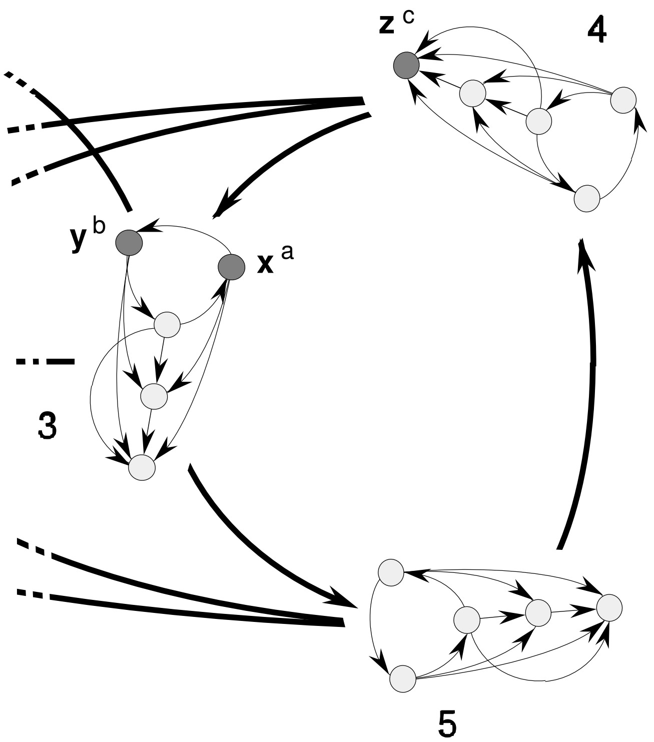

Figure 7: Start state of examples.

Since T^n is a tournament, *r2* implies that at any point in time, all labels are totally ordered. One should notice that these two requirements are easily met by the unbounded implementation, since for any $n-1$ nodes, one can always choose a dominating node in an unbounded total order graph in order to maintain *r1*, and this will never impair *r2* because the graph does not contain cycles.

Let us begin by showing that the labeling operation executions maintain the following two "invariants" 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).

(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 *r2*.

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. As shown in Figure 7, all examples start at a point in time where $\ell_y^{[b]} = 134$,

$\ell_x^{[a]} = 135$, and $\ell_z^{[c]} = 141$, that is, all labels are totally ordered by $\stackrel{\vee}{\prec}$. In the figure, a label such as $\ell_x^{[a]} = 135$, is denoted by shading node 135, and denoting it with the mark x^a.

Example 3.1 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 $\ell_y^{[b+1]}$. It thus moves to the T^2 subgraph 14, following the rule that the node chosen should be the "lowest node dominating all other nodes with labels." This is actually the most basic rule implied by the definition of $\mathcal{L}$. The move to a dominating node is intended to meet *r1*.

Processes y and z can continue for ever to choose $\ell_y^{[b+2]} = 144$, $\ell_z^{[c+2]} = 145$, $\ell_y^{[b+3]} = 143$... (that is, move in the cycle of 14), 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. A T^2 subgraph is a precedence graph able to accommodate 2 labels and no more. Since *num_labels*('14')=2 in $L_x^{[a+1]}$, that is, there are already 2 labels in the T^2 subgraph. By Line 5 of $\mathcal{L}(\ell, i)$, x will move to $\ell_x^{[a+1]} = 151$, and so on... □

The reader can convince himself that following any labeling operation execution $L_z^{[c]}$ by some process z, the above invariants hold. Furthermore, for the set of labels of processes y ($y \neq z$) that were read in $L_z^{[c]}$'s collect operation (denoted *read*$(L_z^{[c]})$), it is the case that

(3)

$$(\forall \ell_y^{[b]} \in read(L_z^{[c]}))(\ell_y^{[b]} \stackrel{\vee}{\prec} \ell_z^{[c]}).$$

This invariant, that the new label chosen is greater than all those read, is the basis for meeting requirement *r1*.

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

Example 3.2 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 choose the same node, they were concurrent, and can be ordered by relative *id*. If any of them were to continue 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 $\ell_z^{[c+1]} = 143$. 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. □

If nodes 1 and 2 did not exist in a T^2 subgraph (that is, each T^2 subgraph was a cycle of 3 nodes), a process' first move into T^k would be onto a node of the cycle. The reader verify

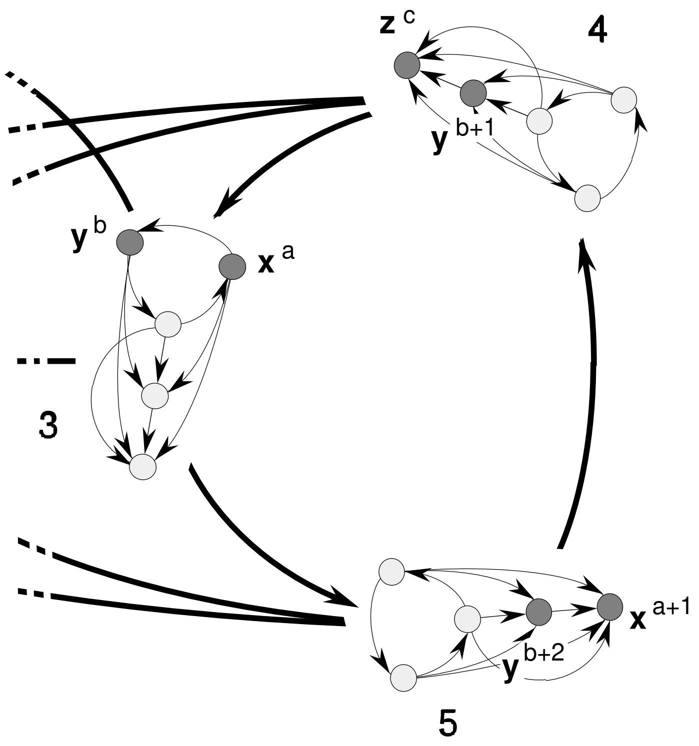

Figure 8: A collect returning a cycle.

that the sequence of operations in Example 3.2, given that T^2 is just a cycle, would cause the labels of x,y and z to end up each on a different node of the cycle, contradicting the first invariant. Based on the existence of nodes 1 and 2, this does not occur.

The following is intended to explain to the reader why for a given level k ($k = 2$ in the example), even if more than k 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 principle*[7]:

> If there are $k+1$ people, each of which first raises a flag, and then counts 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, and 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 3.3, appearing in Figure 8, shows that even though by the above, there are at most k labels at a time in any T^k subgraph, the sets of labels *read* in a labeling operation execution, may contain cycles.

[7]The proof follows since the last person to start counting flags must have seen $k+1$ flags raised.

Example 3.3 Process z begins performing $L_z^{[c+1]}$, reading $\ell_x^{[a]} = 135$. 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 $\ell_x^{[a]} = 135$, $\ell_y^{[b+2]} = 152$, 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, without moving directly into the cycle. However, there is seemingly a problem, since z did not read the label $\ell_x^{[a+1]} = 151$, and so, how can it know that there are already 2 labels in the T^2 subgraph 15? The solution is based on the fact that z can indirectly deduce the existence of $\ell_x^{[a+1]} = 151$. By the first invariant, in all of the cycle of T^3 there are at most 3 labels. 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 a label of x. This rule is maintained by application of Line 4 in $\mathcal{L}^k$.

If the above scenario had occurred in the cycle of a T^k graph, where $k > 3$, then in order to allow the same reasoning as above, it would have to be that z's reading $\ell_y^{[b+2]} = 152$ (or $\ell_y^{[b+2]} \in \{153, 154, 155\}$), would imply that there are $k-2$ labels apart from that of y in the T^{k-1} subgraph 15. It would thus have to be that if $\ell_y^{[b+2]}$, is on supernode 2 in 15, it already established the existence of $k-2$ (and not just 1) other labels in supernode 1.

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. This creates a situation whereby as long as there are $k-1$ or less labels in T^k, all labels enter and move around in supernode 1. Supernode 2 can be chosen in $L_y^{[b+2]}$ only if $k-1$ labels were established by it as being in supernode 1 (i.e. supernode 1 is full). Since supernode 2 is a "bridge," that some process must "cross" (choose) before any process can move into the cycle, the above reasoning for z, in case it read $\ell_y^{[b+2]} \in \{152, 153, 154, 155\}$, holds.

Though the above invariants hold, it follows from Example 3.3 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. Recall that $read(L_y^{[b]})$ denotes the set of label values read in the collect of $L_y^{[b]}$. Let us define the following *observed* relation among labeling operation executions, to be the transitive closure of the *read* relation.

Definition 3.1 *A labeling operation execution $L_x^{[a]}$ is* observed *by $L_y^{[b]}$ (denoted $L_x^{[a]} \xrightarrow{obs} L_y^{[b]}$) if $\ell_x^{[a]} \in read(L_y^{[b]})$ or there exists an $L_z^{[c]}$ such that $\ell_z^{[c]} \in read(L_y^{[b]})$ and $L_x^{[a]} \xrightarrow{obs} L_z^{[c]}$.*

Definition 3.2 *Let the* maximal observed *set $max\_obs(L_x^{[a]})$ be defined as:*

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

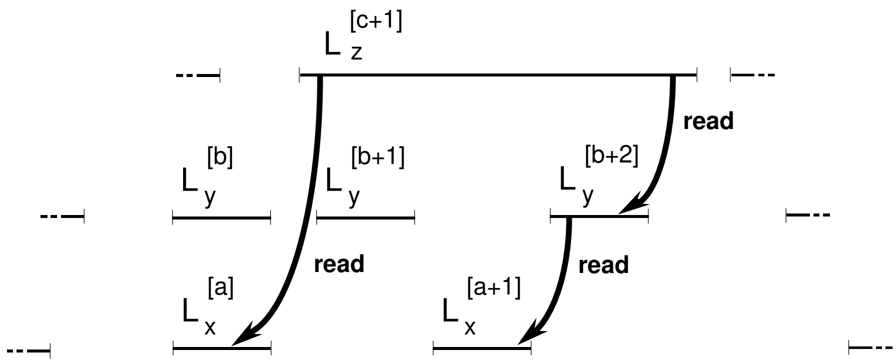

Figure 9: The Observed Relation.

It thus consists of the "latest" of labeling operation executions observed for each process. In a concurrent execution, instead of invariant (3) stating that the new label chosen is greater than all the labels read, it is the case that

(3')

$$(\forall \ell_y^{[b]} \in max\_obs(L_x^{[a]}))(\ell_y^{[b]} \stackrel{\vee}{\prec} \ell_x^{[a]}).$$

The new label chosen is greater than the latest of those observed for each process. As shown in Figure 9, for the labeling $L_z^{[c+1]}$ of Example 3.3, though z read $\ell_x^{[a]} = 143$, and $\ell_z^{[c+1]} \stackrel{\vee}{\prec} \ell_x^{[a]}$, it is the case that its maximal observed label is $\ell_x^{[a+1]}$, and $\ell_x^{[a+1]} \stackrel{\vee}{\prec} \ell_z^{[c+1]}$.

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

Definition 3.3 *Given any two distinct labeling operation executions $L_x^{[a]}$ and $L_y^{[b]}$, $L_x^{[a]} \Longrightarrow L_y^{[b]}$ if either*

1. $L_x^{[a]} \stackrel{obs}{\dashrightarrow} L_y^{[b]}$, *or*

2. $L_x^{[a]} \stackrel{obs}{\not\dashrightarrow} L_y^{[b]}$, $L_y^{[b]} \stackrel{obs}{\not\dashrightarrow} L_x^{[a]}$, *and* $\ell_x^{[a]} \stackrel{\vee}{\prec} \ell_y^{[b]}$.

Since with every $L_x^{[a]}$ there is an associated label $\ell_x^{[a]}$, $\Longrightarrow$ can be seen as a "lexicographical" order on a pairs $(L_x^{[a]}, \ell_x^{[a]})$. The first element in the pair is ordered by $\stackrel{obs}{\dashrightarrow}$, a partial order that is consistent 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 $\stackrel{\vee}{\prec}$, an irreflexive and antisymmetric relation. Parts of the rather involved reasoning as to why the "static" relation $\stackrel{\vee}{\prec}$ on the labels, completes the "dynamic" partial order $\stackrel{obs}{\dashrightarrow}$, to a total order on all labeling operation executions, are provided in Section **??**. The main difficulty is in establishing transitivity. The intuition as to why $\Longrightarrow$ is transitive, is based on the fact that "at any point in time," the current labels of all processes are totally ordered, that is, no three labels are on three different supernodes of a cycle in any T^k subgraph. The reader is encouraged to try to bring about a scenario where there are three labeling operation executions such that

$$L_x^{[a]} \Longrightarrow L_y^{[b]} \Longrightarrow L_z^{[c]} \Longrightarrow L_x^{[a]},$$

while keeping in mind that $\overset{obs}{\dashrightarrow}$ is transitive. It will become clear that this requires that at some point in time, there will be three labels of x, y and z on three different supernodes of a cycle in some T^k subgraph, a contradiction.

References

[AAD*89] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit, Atomic Snapshots of Shared Memory. *Journal of the ACM,* Vol. 40, No. 4 (September 1993), pp. 873–890.

[AB87] U. Abraham and S. Ben-David. Informal and formal correctness proofs for programs (for the critical section problem). 1987. preprint.

[Abr88] K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of 7th PODC*, pages, Toronto, Ontario, Canada, August 1988.

[AD*94] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit. A Bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems*, (16)3, pages 939–953, May 1994.

[A93] Attiya, H., and Rachman, O. Atomic snapshots in O(nlogn) operations. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing,* (Aug. 1993) 29–40.

[AG88] J. Anderson and M. Gauda. The virtue of patience: concurrent programming with and without waiting. 1988.

[And89a] J. H. Anderson. Multiple-writer composite registers. Distributed Computing, Vol. 7, No. 4, pages 175–, 1994.

[Ben88] S. Ben-David. The global time assumption and semantics for concurrent systems. In *Proceedings of the 7^{th} Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada*, pages 223-231, August 1988.

[Blo87] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the 6^{th} Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1987. Revised version to appear in special issue *IEEE Transactions On Computers (Special issue on Parallel and Distributed Algorithms.).*

[BP87] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages, ACM, August 1987.

[CD93] Chandra T. D. and Dwork, C. Using Consensus to solve Atomic Snapshots. Manuscript, 1993.

[CIL87] B. Chor, A. Israeli, and M. Li. Wait-Free Consensus Using Asynchronous Hardware. *SIAM Journal on Computing*, 23, 1994.

[CS91] R. Cori and E. Sopena. Some combinatorial aspects of timestamp systems. Unpublished Manuscript, 1991.

[DS89] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. In *Proceedings of the* 21^{st} *Annual ACM Symposium on Theory of Computing, Seattle, Washington*, pages 454–465, 1989.

[DW92] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or Bounded concurrent timestamp systems are comprehensible! *Proc. 24th ACM Symposium on Theory of Computing (STOC)*, pages 655–666, 1992.

[DPHW92] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time lapse snapshots. *Proceedings of the Israel Symposium on the Theory of Computing and Systems.* Haifa, Israel, pages 154–170, May 1992 (Dolev D., Galil Z., and Rodeh M. eds.).

[DGS88] D. Dolev, E. Gafni, and N. Shavit. Towards a non-atomic era: ℓ-exclusion as a test case. In *Proceedings of the* 20^{th} *Annual ACM Symposium on Theory of Computing*, ACM SIGACT, ACM, 1988.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8:165, 1965.

[FLBB79] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proceedings of 20th FOCS*, pages, October 1979.

[FLBB89] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.

[G92] R. Gawlick. Concurrent timestamping made simple. Masters Thesis, Technical Report MIT/LCS/TR-556, Laboratory for Computer Science, MIT, 1992.

[GLS92] R. Gawlick, N. Lynch, and N. Shavit. Concurrent Time-Stamping Made Simple. In *Proceedings of the Annual Israel Symposium on Theory and Practice of Computing*, Haifa, May 1992.

[Her91] M. P. Herlihy. Wait-free synchronization. In *ACM Transactions on Programming Languages and Systems*, 13(1), pages 124–149, January 1991.

[ICMT94] M. Inoue, W. Chen, T. Masuzawa and N. Tokura. Linear-time Snapshot using Multi-writer Multi-reader registers. *Workshop on Distributed Algorithms*, pages 130-140, Spinger Verlag, 1994.

[HW88] M. P. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, vol. 12, num 3, pages 463–492, July 1990.

[IL93] A. Israeli and M. Li. Bounded time stamps. *Distributed Computing*, (6), 1993.

[IP91] A. Israeli and M. Pinchasov. A linear time bounded concurrent timestamp scheme. Technical Report, Technion, Haifa, Israel, March 1991.

[Kat78] H. Katseff. A new solution to the critical section problem. In *Proceedings of the 10^{th} Annual ACM Symposium on Theory of Computing*, pages, ACM, 1978.

[Lam74] L. Lamport. A new solution of *Dijkstra's* concurrent programming problem. *Communications of the ACM*, 78(8):453–455, 1974.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, (20):806–811, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[Lam86a] Leslie Lamport. The mutual exclusion problem.part i: a theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986.

[Lam86b] Leslie Lamport. The mutual exclusion problem.part ii: statement and solutions. *J. ACM*, 33(2):327–348, 1986.

[Lam86c] Leslie Lamport. On interprocess communication. part i: basic formalism. *Distributed Computing*, 1(12):77–85, 1986.

[Lam86d] Leslie Lamport. On interprocess communication. part ii: algorithms. *Distributed Computing*, 1(12):86–101, 1986.

[LH89] E.A. Lycklama and V. Hadzilacos. A first-come-first-served mutual exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems*, 13(4):558–576, October 1991.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Tecnical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, 1987.

[LTV89] M. Li, J. Tromp, and P. Vitanyi. How to share concurrent waitfree variables. 1989. Unpublished manuscript, Journal Version of LV87.

[LV87] M. Li and P. Vitanyi. *A Very Simple Construction for Atomic Multiwriter Registers*. Report, Aiken Computation Laboratory, Harvard University, 1987.

[New87] R. Newman-Wolfe. A protocol for waitfree atomic multi-reader shared variables. In *Proceedings of the 6^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages, ACM, August 1987.

[PB87] G.L. Peterson and J. Burns. Concurrent reading while writing ii: the multi-writer case. In 28^{th} *Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages, May 1987.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Info. Proc. Lett.*, 12(3):115–116, 1981.

[Pet83] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 1(1):46–55, July 1983.

[Pet88] G. Peterson. Personal communication. 1988.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* North Oxford Academic, England, 1986. Originally published in French in 1984. Translated by D. Beeson.

[SAG94] A. Singh, J. Anderson, and M. Gauda. The elusive atomic register. *Journal of the ACM,* vol. 41, num. 2, pages 311–339, March 1994.

In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages, ACM, August 1987.

[Sch88] R. Schaffer. On the correctness of atomic multi-writer registers. 1988. Bachelor's Thesis, June 1988, Massachusetts Institute Technology. Also, Technical Memo MIT/LCS/TM-364.

[Sha90] N. Shavit. *Concurrent Time-Stamping.* PhD thesis, School of Mathematics and Computer Science, Hebrew University, Jeruslem, 1990.

[SZ91] M. Saks and F. Zaharoglou. Optimal space distributed move-to-front lists. In *Proceedings of the* 10^{th} *Symposium on the Principals of Distributed Computing*, pages 65-73, Montreal, 1991.

[VA86] P. Vitanyi and B. Awerbuch. Shared register access by asynchronous hardware. In 27^{th} *Symp. Found. Computer Sci.*, pages, IEEE, 1986.