UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Lower Bounds For The Space Complexity Of Timestamp Implementations

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
Maestro en Ciencias
(computación)

PRESENTA:
EDUARDO GERÓNIMO PACHECO GÓMEZ

DIRECTORES DE TESIS:
DR. LISA HIGHAM & DR. SERGIO RAJSBAUM

México Enero de 2011

Lower Bounds For The Space Complexity Of Timestamp Implementations

Eduardo Gerónimo Pacheco Gómez

Contents

Chapter 1

Introduction

A distributed system is a collection of independent processors or processes, running at the same time and able to communicate among themselves in order to solve either a common task or a collection of independent tasks [9]. When processes collaborate for solving a common task they have to coordinate their actions. The rise of multi-core machines and the increasing use of computer networks during the last two decades has contributed to the increasing importance of distributed systems. Nowadays distributed systems are everywhere: in academia, bussines, government and home [9]. Many applications require distributed processing. However, there are inherent difficulties introduced by distributed systems. Since each independent processor has just local information, it is impossible to know precisely the absolute or relative time when events happen, also processors may fail unexpectedly which may cause the whole system to malfunction. Also concurrency in these systems gives rise to several difficulties such as inconsistent data, deadlocks, etc. "The explosive growth of distributed systems makes it imperative to understand how to overcome these difficulties" [9]. Timestamps mechanisms have been used to solve many of these problems.

Alan Turing and Alonzo Church established the foundations of sequential computing. Church did it through *lambda calculus* while Turing did it through what we now call *Turing Machines* [22]. These models are considered universal models of sequential computing. They allow us to identify fundamental problems, describe algorithms and identify what kind of problems can be solved and how to measure the efficiency of an algorithm in such systems. In distributed computing there is no one universally accepted model of computation [9]. This is because distributed systems tend to be so different from one another. The main differences among distributed systems are: how processes communicate (through messages or through shared memory), whether processes are synchronous or asynchronous and the kind of failures tolerated. As in the sequential world, some measures to evaluate the efficiency of distributed algorithms are needed. The main complexity measures of interest in distributed systems are: time and space. We define and discuss more about these measures in Chapter 2.

In order to study timestamps, we first state the model we are going to consider. This is important because in distributed computing a small change in the model can radically alter the class of problems that can be solved and their solutions [15]. Through this goal in

Chapter 2, we briefly talk about some distributed system models and discuss some differences among them. Our main goal in that chapter is to describe the model of computation we are going to consider for subsequent chapters.

We study the space complexity of timestamp systems in the typical *shared memory model* in which n asynchronous processes communicate through m read/write atomic registers. This means processes run at different speeds and communicate by calling the methods read and write of the shared registers [15]. Also we assume that each process owns a unique identifier. In such systems, processes have no information about the real time order of events that are incurred by other processes [19]. This causes uncertainty about the order in which operations take place. Reliable information about the relative order in which events take place in such systems is crucial to solve many problems effectively [28]. Timestamps provide such information to processes.

The behavior of a timestamp system is specified by its definition and by progress and correctness conditions. Chapter 2 explains the progress and correctness conditions for timestamp mechanisms that we will consider throughout this thesis. At this point, we just mention that all results we present in this work hold for timestamp implementations that satisfy the *non-deterministic solo termination* property. In other words, our results hold for those implementations in which the progress of a process in the absence of synchronization conflicts is guaranteed.

In a timestamp mechanism, processes get labels or *timestamps*; this labels can be compared to provide the necessary partial information about the order in which events occur in a system [28]. Timestamps have been used to solve some fundamental problems in distributed computing, for example: consensus[1], registers constructions [20, 27, 34], snapshot algorithms [2], adaptive renaming algorithms [8] and k-exclusion algorithms [3, 17, 26, 33].

According to the size of timestamps provided to processes, timestamps mechanisms may be either *bounded* or *unbounded*. In a *bounded timestamp* mechanism, processes are able to get timestamps whose size is bounded by a function of the total number of processes in the system while in an unbounded one the size of a timestamp is not bounded. In many theoretical and practical works, bounded solutions are preferred over the unbounded ones. However, there are examples where unbounded timestamps are used [26, 34]. In this thesis we are concerned principally with the necessary number of registers to implement an unbounded timestamp mechanism. In the first part of this work, we assume a timestamp mechanism that allows processes to get a timestamp more than once. Such kind of mechanisms are known as *long-lived*. Later on, we give a formal definition of a long-lived timestamp system.

Sometimes in distributed computing if a problem can not be solved under certain constraints we can change or vary those constraints and consider a slightly different version of the problem. For instance, we can restrict the number of times a process is allowed to use an object to one. An object is *long-lived* if processes can repeatedly use it, and it is *one-shot* if it is available only once. In some cases one-shot object implementations are simpler than long-lived ones. Such is the case for renaming, mutual exclusion, splitters and snapshot objects. On the other hand, some problems are inherently one-shot, such as consensus or non-resettable test and set objects. In Chapter 4, we study the space complexity of one-shot unbounded timestamp implementations.

The main results are presented in Chapter 3 and Chapter 4. These results were obtained during my research stay at the University of Calgary under the supervision of Dr. Lisa Higham and Dr. Philipp Woelfel. Our results are: a wait-free implementation of an unbounded one-shot timestamp mechanism and lower bounds on the necessary number of read/write atomic registers for implementing any unbounded long-lived and one-shot timestamp system. Regarding the lower bounds, we obtained two new results. First, in Chapter 3, we present a linear lower bound on the number of registers needed to implement a long-lived and unbounded timestamping system. This lower bound is an improvement to the one presented in [14] by Faith Ellen, Panagiota Fatourou and Eric Ruppert who proved an $\Omega(\sqrt{n})$ lower bound. Our second result concerns the space complexity of one-shot timestamp implementations. In Chapter 4, we prove an $\Omega(\sqrt{n})$ lower bound for any one-shot unbounded timestamp implementation. As far we are aware, there are no previous results regarding one-shot unbounded timestamp systems. The wait-free implementation of a one-shot timestamp scheme uses only $n/2$ multi-writer registers. All these results appeared first in [28], so the content of this work is an extension of that paper.

Chapter 5 describes some open questions and some discussion for future research. Some of those questions are related to the implementation of timestamp schemes using strong objects such as `CAS` registers. Other open questions concern the relationship between the timestamp problem and the *Lattice Agreement problem*.

We hope you have some fun as you read the rest of this work.

Chapter 2

Preliminaries

In this chapter, we describe and compare some models of distributed systems in order to give a general view of distributed computing and to highlight the importance of stating precisely the model of a distributed system. Then, we present the model in which our results hold and some preliminaries before stating the timestamp problem.

Based on communication medium and degree of synchrony, three models of distributed systems are mainly studied in the literature: *asynchronous* and *synchronous message-passing* and *asynchronous shared memory*. A system is *synchronous* if all processes take steps at exactly the same speed, otherwise it is *asynchronous*. Asynchrony can be modeled by an adversary that chooses the order in which processes take steps [4, 14]. Processes can communicate among themselves by sending messages to one another or through performing operations on shared data structures or objects. Commonly, it is assumed systems are *eponymous*. In a eponymous system each process has a unique name or identifier. A system, where processes do not have unique identifiers is said to be *anonymous*. An algorithm for a process consists of a local sequential program that determines its state. Then, processes can be modeled as state machines [29, 32]. In distributed systems faulty processes may be specified. We say a process fails, if it stops at some point without any warning.

2.1 Distributed System Models

In a message-passing system, without failures, processes communicate with each other by sending messages over communication channels, where each channel provides a bidirectional connection between two specific processes. Connections that link processes with one another define the topology of the system. The topology may be represented by an undirected graph in which each node represents a process and there exists an edge between two nodes if and only if there is a channel between the corresponding processes. Unidirectional channels are represented by directed edges. A correct channel behaves as a **FIFO** queue. The sender enqueues messages in the channel and the receiver dequeues them. A formal definition of a message-passing system can be found in [9]. This description of message-passing systems gives us an idea about the necessary things to look at when we design algorithms for such systems. Not less important is to know how to measure the efficiency of an algorithm in a message-passing system.

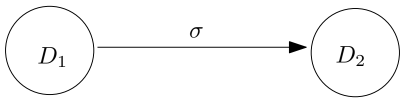
The *message complexity* of an algorithm for either a synchronous or an asynchronous message-passing system is the total number of messages sent [15] by the algorithm. The step complexity of a synchronous message-passing algorithm is the maximum number of rounds in any possible execution of the algorithm, until the algorithm has terminated where a *round* is any sequence of steps such that every process in the system participates at least once. In an asyncrononous system we assume that the maximum message delay in any execution is one unit of time and therefore the time complexity is the maximum time until termination among all possible executions.

Asynchronous and synchronous message-passing systems differ in their set of solvable problems. For example, the *consensus problem*, a fundamental problem in distributed computing [22], can be solved in the presence of failures in synchronous message-passing systems but not in asynchronous ones, even if only one process fails. An alternative to the deterministic model is one in which processes have access to some source of random information such as that provided by flipping a coin or rolling dice [9]. Randomization is a powerful tool used to design distributed algorithms and it allows us to solve problems in situations where they can not be solved deterministically and often makes possible simpler solutions even when deterministic ones exist. In randomized algorithms a process may have many choices for its next step, but the choice is made according to some probability distribution. Termination of randomized algorithms is generally required only with high probability and one considers worst-case expected time rather than worst-case time. Some problems, where randomization helps to overcome impossibility results and lower bounds are: leader-election, mutual exclusion and consensus [4, 21].

There are many other models we can consider, for instance, the *semi-synchronous model*, where processes may run at different speeds but there are bounds on the relative speeds of processes and the *parallel random-access machine* commonly used to study synchronous shared memory systems [15]. However, for the rest of this work we will only consider the asynchronous *shared memory model* that we explain more carefully in the next section.

2.2 The Asynchronous Shared Memory Model

An asynchronous distributed system in the shared memory model is a collection of n processes, $\mathcal{P} = \{p_1, \ldots, p_n\}$, that run concurrently. Each process executes an algorithm and can communicate with other processes. Processes communicate with one another by performing operations on shared data structures or objects. These objects can be of various types. The type of an object specifies the operations that can be performed on the object, the values returned by the operations and the set of possible states for an object of that type. At any time, an object has a state and when a processor performs an operation on it, the object can change to a new state and return a response to the process. An object type is *deterministic* if the outcome of each operation with specified input parameters is uniquely determined by



$$\begin{tikzcd} D_1 \arrow[r, "\sigma"] & D_2 \end{tikzcd}$$

Figure 2.1: Execution

the object's current state and it is *non-deterministic* if more than one outcome is possible for an operation in some states.

An object of type `register` is able to store a value. Its set of atomic operations (or methods) is {`read`, `write`}, such that, each `write` invocation updates the current value stored into the `register` and each `read` invocation returns the value of the last recently write to the register. For the purposes of this thesis, we assume processes communicate only through a set of m objects of type `register` denoted $\mathcal{R} = \{r_1, \ldots, r_m\}$.

There are restricted types of `register`. According to their access pattern they can be: `SWSR register` (single-writer single-reader), `SWMR register` (single-writer multiple-reader) or `MRMW register` (multiple-writer multiple-reader). All of them provide the same set of atomic operations. Examples of types that support more powerful operations are: `read-modify-write register`, `test&set register` and `compare&swap register`. In Chapter 5, we talk more about these objects. The type of the shared objects used for communication determines the class of problems that can be solved and their complexity.

The state of the system is described through configurations. A *configuration* C is a tuple of $(s_1, \ldots, s_n, v_1, \ldots, v_m)$, denoting that process p_i, $1 \leq i \leq n$, is in state s_i, and register r_j, $1 \leq j \leq m$, has value v_j. In an initial configuration, denoted C^*, all processes are in their initial states and all registers contain initial values. A *schedule* σ is a (possibly infinite) sequence of processes indices. We denote the empty schedule by ε. A step is an atomic action which consists of simultaneous changes to the state of some process and the value of some shared register. An *execution* $(C; \sigma)$ is a sequence of steps beginning in configuration C and moving through successive configurations one at a time. At each step, the next process p_i indicated in the schedule σ, takes the next step in its program. If σ is finite, the final configuration of the execution $(C; \sigma)$ is denoted $\sigma(C)$. A configuration C is *reachable from a configuration* D if there exists a finite schedule, σ, such that $\sigma(D) = C$, and it is just *reachable* if $\sigma(C^*) = C$. If σ and π are finite schedules then $\sigma\pi$ denotes the concatenation of σ and π. Let P be a set of processes, and σ a schedule, we say σ is *P-only* if only indices of processes in P appear in σ.

We will use directed graphs to depict configurations and executions. A node represents a configuration and directed edges will be labeled with schedules. A directed graph as in Figure 2.1 represents the execution $(D_1; \sigma)$, where $D_2 = \sigma(D_1)$.

Any execution $(C; \sigma)$ defines a partial *happens before* order "$\rightarrow$" on the method calls that occur during $(C; \sigma)$. A method call m_1 happens before m_2, denoted $m_1 \rightarrow m_2$, if the response of m_1 occurs before the invocation of m_2. Two configurations $C_1 = (s_1, \ldots, s_n, r_1, \ldots, r_m)$

and $C_2 = (s'_1, \ldots, s'_n, r'_1, \ldots, r'_m)$ are *indistinguishable* to process p_i denoted $C_1 \sim_{p_i} C_2$ if $s_i = s'_i$ and $r_j = r'_j$ for $1 \leq j \leq n$. If S is a set of processes, and for every process $p \in S$, C_1 and C_2 are indistinguishable to p, then we say C_1 and C_2 are indistinguishable for S denoted $C_1 \sim_S C_2$. Note that if $C_1 \sim_S C_2$ then for any S-only schedule σ, $\sigma(C_1)$ and $\sigma(C_2)$ are indistinguishable to S. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the set of processes in the system. Then, for $1 \leq k \leq n$, we define $\mathcal{P}_k = \{p_1, \ldots, p_k\} \subseteq \mathcal{P}$, this notation will be very useful for proofs in Chapter 3.

Proofs in Chapter 3 and Chapter 4 use covering arguments. We say process p_i *covers* register r_j in a configuration C, if the one step execution $\big(C; i\big)$ is a write to register r_j. A set of processes P covers a set of registers R if for every register $r \in R$ there is a process $p \in P$ such that p covers r. A *block-write* by P to R is an execution $(C; \pi)$, where π is a permutation of P. For a process set P, we denote by π_P an arbitrary (but fixed) permutation of P (for example the one that orders processes by their ID). Thus, if C is a configuration in which P covers a set of registers R, then $(C; \pi_P)$ is a block-write by P to R.

The amount of shared memory used to solve a problem by a distributed algorithm A is what we call the *space complexity* of A. We can state the space complexity of a distributed algorithm A algorithm according to the number of different shared registers required by it, or by the amount of shared space that it used in bits. A different way to assess the efficiency of an algorithm to solve a problem P can be made by counting the number of steps taken by the processes in the system to P in the worst case: the *step complexity* of the algorithm. The results presented in Chapter 3 and Chapter 4 concern the space complexity for implementing timestamping mechanisms. Before stating the timestamp problem we discuss the safety and progress conditions that allows us to specify the adequate behavior of timestamp objects.

2.3 Safety and Progress Properties

We are concerned about correct implementations of concurrent timestamping objects in shared memory systems. But what does it mean for a concurrent object implementation to be correct? Informally, an object implementation is correct if it behaves adequately. *Safety* and *liveness* properties specify how concurrent objects or systems should behave. These properties specifies what guarantees are provided by the system through stating the behavior of the objects that compose it. A safety property states that *something bad* in the system never happens, while a liveness property states that eventually *something good* happens. One example of a safety property is *linearizability* [23]. Linearizability provides the illusion that each operation applied on a concurrent object takes effect instantaneously at some point between its invocation and its response. Other safety properties, also known as *correctness* conditions, are *sequential consistency* and *quiescent consistency*. For a formal definition of these properties see [22].

Examples of liveness properties (also known as progress conditions) are: *wait-free* and *lock-free*, both of these properties guarantee that the delay of a process operating on an object does not prevent others from taking steps. A concurrent object is *wait-free* if it guarantees every non-faulty process will complete its operation within a finite number of its own steps and it is *lock-free* if at least one non-faulty process will complete its operation in a finite number of its own steps. These two progress conditions can be extended to randomized wait-freedom and randomized lock-free by considering the expected number of steps. In this thesis we consider a strictly weaker progress condition for timestamp implementations: *non-deterministic solo-termination*. An implementation has the *non-deterministic solo-termination* property if, for every configuration C and every process p_i, there exists a $\{p_i\}$-only execution, starting at configuration C, in which p_i finishes executing its operations. In other words, if p_i runs long enough without encountering a synchronization conflict, it will make progress. The last property is equivalent to *obstruction-free* for deterministic algorithms [14].

2.4 Timestamps

In an asynchronous distributed system is impossible for processes to determine the exact temporal ordering of all events [14], where events correspond to method invocations and responses. Having partial information about this ordering is crucial to solve many problems effectively. A Timestamp mechanism helps processes achieve this. It allows processes to label events and compare those labels to obtain information about the real time order in which the corresponding events took place. Timestamps mechanisms result in an extremely powerful tool for concurrency control. Timestamps are used in many areas of computer science and in many practical applications. Some examples, as we mentioned in Chapter 1, are consensus [1], registers constructions [20, 27, 34], snapshot algorithms [2], adaptive renaming algorithms [8] and k-exclusion algorithms [3, 17, 26, 33].

The history of timestamps began with Leslie Lamport in 1978. He was the first one to devise a timestamp mechanism [26]. To do so, he defined a *happens before* relation on events occurring in message-passing systems to reflect the causal relationship of events. This *happens before* relation is a partial order, where, informally, an event e_1 happens before event e_2, if e_1 can cause or influence e_2. Then, Lamport devised a *logical clock* that assigns an integer value $C(e)$, called a *timestamp*, to each event e such that if event e_1 happens before event e_2 then $C(e_1) < C(e_2)$. The Lamport's logical clock system based on integers was later extended to clocks based on vectors by Collin Fidge [16] and Friedemann Mattern [30]. Lamport also devised a very simple shared memory timestamp mechanism [26] that uses n `SWMR registers`, where n is the number of processes in the system. When a process gets a timestamp, it collects all the values in all the n registers and writes one plus the maximum value it read into its corresponding register. Such a value is returned as timestamp. Cinthya Dwork and Orli Waarts [13] devised a different implementation, where timestamps are vectors and they are compared lexicographically. Hagit Attiya and Arie Fouren proposed a more complicated implementation in [8]. Rachid Guerraoui and Eric Ruppert [19] considered timestamps in anonymous systems (recall that in an anonymous system processes do not have unique identifiers). They devised a wait-free timestamp implementation for such systems. The construction of Hagit Attiya and Arien Fouren uses an unbounded number of

`MWMR registers`, while Rachid Guerraoui and Eric Ruppert assume n `MWMR registers`.

In all implementations mentioned above the size of timestamps is not bounded by a function of n, the total number of processes, and they assume processes are allowed to get a timestamp many times. Finding such an implementation is known as the *unbounded timestamp problem*. On the other hand, finding a timestamp implementation, where the size of timestamps is bounded by a function of n is called the *bounded timestamp problem*. We describe these two versions of the problem in more detail in the next two subsections.

2.4.1 Bounded Timestamps

Amos Israeli and Ming Li were the first to isolate the notion of bounded timestamping as an independent concept. They developed in [24] a theory of bounded sequential timestamping systems. Danny Dolev and Nir Shavit were the first to present a bounded construction of a concurrent timestamping system [11]. They achieved so by transforming an unbounded solution into a bounded one. Cynthia Dwork and Orli Waarts [12], Rainer Gawlick *et.al.* [18], Amos Israeli and Ming Li [24], Amos Israeli and Meir Pinhasov [25] followed the same strategy. In this chapter, we introduce formally the bounded timestamp problem. We do not describe any particular implementation since they are quite complex and it is out of the scope of this work.

We follow the presentation of the bounded timestamp problem that appears in [13]. We assume an asynchronous system of n concurrent processes that communicate among themselves through n read/write atomic `SWMR` registers of bounded size. A bounded timestamping system must support two wait-free operations: `label` and `scan`. The first operation assigns an element(label) from a finite timestamp universe $\mathcal{T}$ to the process that performs it. Every process is allowed to perform any number of `label` operations. For $1 \leq i \leq n$, let L_i^k denote the kth `label` operation performed by process i (the superscript k is not visible to process i, this is simply a notational device for describing long-lived runs of the timestamp system). Analogously l_i^k denotes the label obtained by process i during L_i^k. A `scan` operation returns a pair $(\bar{l}, \prec)$ where $\bar{l}$ is an indexed set of labels (one per process) $\{l_1, \ldots, l_n\}$ and $\prec$ is an irreflexive total order among the elements of $\bar{l}$. We denote the kth `scan` operation performed by process i by S_i^k. In order to handle initial conditions, we assume that each process i has an initial timestamp $\perp \in \mathcal{T}$ denoted l_i^0. To avoid making any distinction between initial timestamps and the timestamps assigned by `label` operations, we say that timestamp l_i^0 was returned to process i by a fictitious initial timestamping operation L_i^0 that took place just before the beginning of the execution and we assume all of them are concurrent.

Each process' program consists of a sequence of `label` and `scan` operations which are totally ordered by the precedence order $\rightarrow$ (recall that if A and B are two different operations then $A \rightarrow B$ if B starts after A has terminated). This means the operations of any process are not concurrent. Furthermore, three properties are required:

1. Ordering. There exists an irreflexive total order $\Longrightarrow$ on the set of all `label` operations, such that:

- Precedence: For any pair of `label` operations L_i^a and L_j^b (where possibly $i = j$) if $L_i^a \rightarrow L_j^b$, then $L_i^a \Longrightarrow L_j^b$.
- Consistency: For any `scan` operation S_i^k returning $(\bar{l}, \prec)$, for all l_i^a and l_j^b in $\bar{l}$, $l_i^a \prec l_j^b$ if and only if $L_i^a \Longrightarrow L_j^b$.

2. Regularity. For any timestamp l_i^a in $\bar{l}$ returned by S_p^k, L_i^a begins before S_p^k terminates, and there is no L_j^b such that $L_i^a \rightarrow L_j^b \rightarrow S_p^k$.

3. Monotonicity. Let S_i^k and $S_j^{k'}$ (where i and j may be equal) be a pair of `scan` operations returning the indexed sets $\bar{l}$ and $\bar{l}'$ respectively which contain labels l_p^a and l_p^b respectively. If $S_i^k \rightarrow S_j^{k'}$ then $a \leq b$.

Intuitively, the ordering property says that there exists a total order of the `label` operations which is consistent with the actual order of events. This order is not necessarily known by the processes. The regularity condition establishes that the timestamps returned by an `scan` operation are not obsolete. Monotonicity says that if an `scan` operation preeceeds another one and the first one returns a new timestamp for some process p, then the second can not return an older timestamp for p.

Implementations of bounded timestamps systems are tricky because of the need to recycle labels. Hence the core of the known implementations is to construct a mechanism that helps to reuse labels. Some implementations may be found in [12, 18, 25].

2.4.2 Unbounded Timestamps

An unbounded timestamp system provides two algorithms for each process: `getTS` and `compare`. The first one takes no arguments and returns a value from an infinite timestamp universe $\mathcal{T}$. The `compare` algorithm takes two arguments from $\mathcal{T}$ and outputs a boolean value. The happens before relation orders time intervals associated with method calls (recall that method call m_1 *happens before* method call m_2, if the response of m_1 precedes the invocation of m_2). If a `getTS` method g_1 that returns t_1 happens before another `getTS` method g_2 that returns t_2 then any later `compare`(t_1, t_2) must return true and any later `compare`(t_2, t_1) must return false. Two concurrent instances of `getTS` are allowed to output the same timestamp while non-concurrent instances are not. There are stronger versions of bounded timestamps systems, for instance those that are *static*. In a static timestamp implementation, for every pair t and t', the `compare`(t,t') always outputs the same result in all executions. Thus, in such implementations a `compare` method does not require access to shared memory.

In a linearizable timestamp implementation each invocation of `getTS` and `compare` must appear to take effect atomically and in some order consistent with real-time order. If one such operation instance ends before another begins, then the former must be linearized before the latter. If g_1 and g_2 are concurrent instances of `getTS`, then one of them must be linearized before the other, so they can not receive the same or incomparable timestamps. We do not consider linearizable timestamps implementations. Therefore, concurrent `getTS` method calls can output incomparable timestamps. However, that is not a big constraint since in eponymous systems there is a way to get a linearizable timestamping system from those that are not. If we attach processes id's to granted timestamps [14], the order of the id's establishes the corresponding linearization points and concurrent method calls of `getTS` would return different labels since all processes own a different id. Notice that in anonymous systems this can not be achieved.

We say a timestamp system is long-lived if processes are allowed to call the method `getTS` many times, while it is said to be one-shot if processes are allowed to invoke the method `getTS` only once. The `compare` method in both versions can be invoked any number of times.

Often, $\mathcal{T}$ is a partially ordered set and all timestamps returned by `getTS` method calls during an execution are comparable and preserve the happens before relation of these method calls. Non-static timestamp objects can lead to different partial orders of the set $\mathcal{T}$.

As far we are aware, the only paper that studies the space complexity of unbounded timestamp implementations is due to Faith Ellen, Paniagota Fatourou, and Eric Ruppert [14]. They studied the number of atomic registers needed to implement any long-lived unbounded timestamp mechanism. Faith Ellen *et.al.* showed that any implementation of a long-lived unbounded timestamp mechanism that satisfies non-deterministic solo termination requires at least $\frac{1}{2}\sqrt{n-1}$ `MWMR registers`, where n is the number of processes in the system. This lower bound holds for non-static implementations and even for those that are static. In the same paper they proved that if only `SWMR registers` are available then $n-1$ is optimal. They proved the upper bound by showing an algorithm that uses such a number of registers.

Furthermore, they proved $\Omega(n)$ registers are required by any timestamp implementation that satisfies non-deterministic solo termination, where $\mathcal{T}$ is a nowhere dense partially ordered universe, exactly matching known implementations. A partially ordered set $\mathcal{U}$ is called nowhere dense if, for every two elements $x, y \in \mathcal{U}$, there are only a finite number of elements $z \in \mathcal{U}$ such that $x < z < y$. They also prove matching upper and lower bounds for anonymous systems. They show a wait-free timestamp mechanism that uses $O(n)$ `MWMR registers` for anonymous systems. In such implementation $O(n^3)$ steps are necessary for a process to get a timestamp.

The most general lower bound proved in [14] shows that there is a big gap between lower and upper bounds for the space complexity of timestamp implementations. In the next chapter, we prove a new lower bound that substantially reduces such a gap. In Chapter 3, we prove that any non-deterministic solo termination and long-lived unbounded timestamp implementation uses at least $n/6 - O(1)$ `MWMR registers`.

Chapter 3

Space Lower Bound For Long-Lived Timestamps

In this chapter we prove a new lower bound for any long-lived timestamp implementation that satisfies non-deterministic solo termination. We prove that a linear number of read/write atomic registers is needed to implement such systems. Recall that a timestamp object is long-lived if each process is allowed to invoke the `getTS` method many times (the `compare` method can be invoked many times even in the one-shot version of the problem). To prove our lower bound we strongly rely on a previous result due to Ellen, Fatourou and Ruppert [14] which we restate here using a convenient notation for this work.

The proofs we present in this chapter form part of the first result that we obtained in a research project regarding bounded timestamps at the University of Calgary under the supervision of Dr. Lisa Higham and Dr. Philipp Woelfel. Therefore, original proofs can be found in [28].

Lemma 1. *Consider any timestamp implementation from registers that satisfies non-deterministic solo termination and let C be a reachable configuration. Let B_0, B_1, B_2, A_0, A_1 be disjoint sets of processes, where in C each of B_0, B_1, and B_2 cover a non-empty set R of registers. Then, there exists $i \in \{0, 1\}$ such that every A_i-only execution starting from $C_i = \pi_{B_i}(C)$ that contains a complete* `getTS()` *operation writes to some register not in R.*

Proof. Let B_0, B_1, B_2, A_0 and A_1 be disjoint sets of processes as in the statement of Lemma 1 and σ_i be any A_i-only schedule, $i \in \{0, 1\}$, such that the execution $(C_i; \sigma_i)$ contains a complete `getTS` instance. Recall that C_i denotes the resulting configuration after B_i performed a block write to R in C and π_{B_i} denotes an arbitrary but fixed permutation of B_i. Let t_i be the timestamp granted in $(C_i; \sigma_i)$. Assume that there is no write outside of R that occurred in either execution $(C_0; \sigma_0)$ or in $(C_1; \sigma_1)$. Then configurations $\pi_{B_0}\sigma_0\pi_{B_1}\sigma_1\pi_{B_2}(C)$ and $\pi_{B_1}\sigma_1\pi_{B_0}\sigma_0\pi_{B_2}(C)$ are indistinguishable to all processes. This is because all information written by processes in A_i to R was obliterated by the last block write performed by B_2. Also note $\pi_{B_i}(C)$ and $\pi_{B_{1-i}}\sigma_{1-i}\pi_{B_i}(C)$ are indistinguishable to A_i. Hence, the value of t_i in $\pi_{B_i}\sigma_i(C)$ and in $\pi_{B_{1-i}}\sigma_{1-i}\pi_{B_i}\sigma_i(C)$ coincide. Furthermore, in $(C; \pi_{B_0}\sigma_0\pi_{B_1}\sigma_1)$ we determine that $t_0 < t_1$ but in $(C; \pi_{B_1}\sigma_1\pi_{B_0}\sigma_0)$ we have $t_1 < t_0$; notice that an invocation

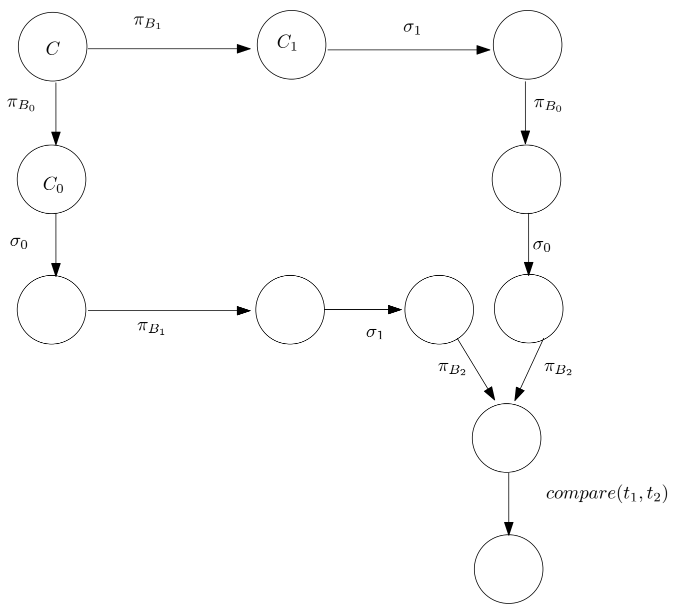


Figure 3.1: Illustration of Lemma 1

to `compare`(t_0, t_1) after the block write by B_2 returns either true or false. This results in a contradiction and therefore either in computation $(C; \sigma_0)$ or in computation $(C; \sigma_1)$ a write to the registers outside of R is performed. □

Lemma 1 and its illustration are taken from [14]. However, the version of the lemma that we presented here is a particular case of the one presented in [14]. Ellen *et.al.* used the general case of Lemma 1 to reach a configuration in which all registers are covered by three processes each. Starting from a configuration where each register in R, a set of registers, is covered by many processes, they invoke Lemma 1 to reach a configuration where a register $r \notin R$ gets covered. Then, they get a new set of registers $R := R \cup \{r\}$ where each register is covered by many processes. This strategy can be repeated if each register in R is covered by at least three processes. In the repeated use of Lemma 1, processes have to be able to invoke the `getTS` method more than once. This is important since such a property implies that their proof holds only for long-lived timestamp implementations.

For our lower bound we proceed in a different way. We increase the number of processes covering some register but prevent registers from being covered by more than three processes. A $(3, k)$-*configuration* is a configuration where k processes are covering registers, but each register is covered by at most three processes. We argue that if from any quiescent configuration there exists a schedule that leads to some $(3, k)$-configuration, we can find an execution (possibly very long), in which at least two $(3, k)$-configurations, let us say C_0 and C_1, are encountered such that in both of them, each register is covered by the same number of processes. We then argue that these configurations can be linked together through a schedule σ. The execution $(C_0; \sigma)$ that leads from C_0 to C_1 starts with three block-writes to the registers that are covered by three processes each. We then apply Lemma 1 to see that we can insert a solo-execution of some unused process p into the schedule σ after one

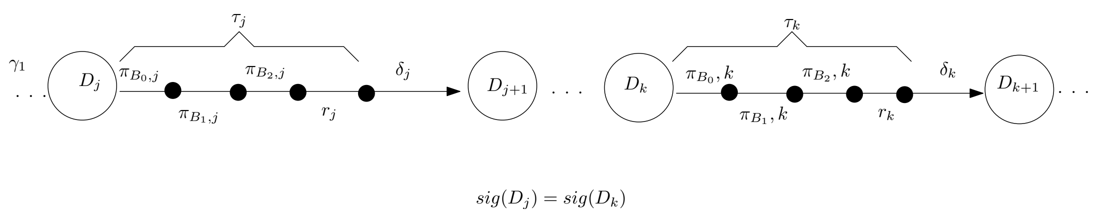


Figure 3.2: Illustration of Lemma 2

of the block-writes, such that at the end of this new execution $(C_0; \sigma')$ process p is about to write outside of the registers that are 3-covered in C_0. Since the other two block-writes are overwriting p's trace in σ', no process (other than p) can distinguish between $(C_0; \sigma')$ and $(C_0; \sigma) = C_1$. It follows that in $\sigma'(C_0)$ process p covers a register that was covered by at most 2 other processes. Hence, we have obtained a $(3, k+1)$-configuration. We can do this for $k \leq \lceil (n-1)/2 \rceil$, so at the end we obtain a $(3, \lceil (n-1)/2 \rceil)$-configuration.

Before presenting our first result we need some definitions. The *signature* of a configuration C, denoted sig(C), is a tuple $(c_1, c_2, \ldots, c_m)$ where every c_i is the number of processes covering the i-th register in C. The set of registers whose corresponding entry in sig(C) is equal to 3 is denoted $\mathcal{R}_3(C)$. A configuration C is a $(3,k)$-*configuration* if sig$(C) = (c_1, c_2, \ldots, c_m)$ satisfies $\sum_{i=1}^{m} c_i = k$ and $c_i \leq 3$ for every $1 \leq i \leq m$. Notice that in any $(3,k)$-configuration there are at least $\lceil k/3 \rceil$ registers covered.

Lemma 2. *Let P be an arbitrary set of processes. Suppose for every quiescent configuration C there exists a P-only schedule σ such that $\sigma(C)$ is a $(3,k)$-configuration. Then for any quiescent configuration D, there are two $(3,k)$-configurations C_0 and C_1, and P-only schedules γ_1, γ_2, and η such that:*

(a) $\gamma_1(D) = C_0$,

(b) $\gamma_2(C_0) = C_1$,

(c) sig$(C_0) =$ sig(C_1), *and*

(d) $\gamma_2 = \pi_{B_0}\pi_{B_1}\pi_{B_2}\eta$, *where B_0, B_1 and B_2 are disjoint sets of processes each covering $\mathcal{R}_3(C_0)$.*

Proof. We inductively define an infinite sequence of schedules $\tau_0, \delta_0, \tau_1, \delta_1, \ldots, \tau_{i+1}, \delta_{i+1}, \ldots$ and reachable configurations $D_0 = D, D_1, D_2, \ldots$, where $D_{i+1} = \tau_i\delta_i(D_i)$, as follows: τ_{i+1} is the concatenation of a sequence of permutations $\pi_{B_{0,i+1}}\pi_{B_{1,i+1}}\pi_{B_{2,i+1}}$ and some P-only schedule r_{i+1} in which every process in P finishes any pending operation; $B_{0,i+1}, B_{1,i+1}$ and $B_{2,i+1}$ are disjoint sets of processes each covering $\mathcal{R}_3(D_{i+1})$. Execution $(D_{i+1}; \pi_{B_{0,i+1}}\pi_{B_{1,i+1}}\pi_{B_{2,i+1}})$ consists of three consecutive block-writes to $\mathcal{R}_3(D_{i+1})$ by the processes in $B_{0,i+1}$, $B_{1,i+1}$, and $B_{2,i+1}$, respectively.

Thus, configuration $\pi_{B_{0,i+1}}\pi_{B_{1,i+1}}\pi_{B_{2,i+1}}r_{i+1}(D_{i+1})$ is quiescent. Schedule δ_{i+1} is chosen such that $\tau_{i+1}\delta_{i+1}(D_{i+1})$ is a $(3,k)$-configuration. By the inductive construction, $\tau_{i+1}(D_{i+1})$

is quiescent and by the hypothesis δ_{i+1} exists.

Since the set of signatures is finite, there are two indices $j < k$, such that $\text{sig}(D_j) = \text{sig}(D_k)$. Fix two such indices j and k. Let $\gamma_1 = \tau_0\delta_0\tau_1\delta_1\tau_2\delta_2\ldots\tau_{j-1}\delta_{j-1}$ and $\gamma_2 = \tau_j\delta$ where $\delta = \delta_j\tau_{j+1}\delta_{j+1}\ldots\tau_{k-1}\delta_{k-1}$. Furthermore, $C_0 = \gamma_1(D)$ and $C_1 = \gamma_2(C_0)$. By definition, the configurations C_0 and C_1 satisfy (a) and (b). Moreover, by construction $C_0 = D_j$ and $C_1 = D_k$ and since $\text{sig}(D_j) = \text{sig}(D_k)$, (c) is satisfied. Finally, Let $\eta = r_j\delta$. Then, $\gamma_2 = \pi_{B_{0,j}}\pi_{B_{1,j}}\pi_{B_{2,j}}\eta$, where $B_{0,j}, B_{1,j}, B_{2,j}$ are disjoint sets of processes each covering $\mathcal{R}_3(D_j) = \mathcal{R}_3(C_0)$. This proves (d). Figure 3.2 illustrates this lemma. □

Recall that if $\mathcal{P} = \{p_1, \ldots, p_n\}$ then $\mathcal{P}_k = \{p_1, \ldots, p_k\} \subseteq \mathcal{P}$ and $P_0 = \emptyset$.

Lemma 3. *For every $0 \leq k \leq \lceil (n-1)/2 \rceil$ and for every quiescent configuration D, there exists a $\mathcal{P}_{2k}$-only schedule σ_k such that $\sigma_k(D)$ is a $(3,k)$-configuration.*

Proof. The proof is by induction on k. For $k = 0$ the claim is obvious.

Let $k \geq 1$, and let D be an arbitrary quiescent configuration. By the induction hypothesis, for every quiescent configuration C, there exists a $\mathcal{P}_{2k-2}$-only schedule σ_{k-1}, such that $\sigma_{k-1}(C)$ is a $(3, k-1)$-configuration. Lemma 2 holds for any P, in particular when $P = \mathcal{P}_{2k-2}$. Hence, by Lemma 2 and the induction hypothesis there are two reachable configurations C_0 and C_1, and $\mathcal{P}_{2k-2}$-only schedules γ_1, γ_2, and η, such that $\gamma_1(D) = C_0$, $\gamma_2(C_0) = C_1$, $\text{sig}(C_0) = \text{sig}(C_1)$, and $\gamma_2 = \pi_{B_0}\pi_{B_1}\pi_{B_2}\eta$, where B_0, B_1 and B_2 are disjoint sets of processes, each covering $\mathcal{R}_3(C_0)$.

Consider the processes $p_{2k-1}, p_{2k} \in \mathcal{P}_{2k} - \mathcal{P}_{2k-2}$. For $i \in \{0,1\}$, let α_i be a $\{p_{2k-i}\}$-only schedule starting in $\pi_{B_i}(C_0)$, in which p_{2k-i} performs a complete `getTS()` instance. According to Lemma 1, there exists $i \in \{0,1\}$, such that p_{2k-i} writes to some register not in $\mathcal{R}_3(C_0)$ during its solo-execution $(\pi_{B_i}(C_0); \alpha_i)$. (Note that whether $i = 0$ or $i = 1$ depends on C_0.) Let r be the first register not in $\mathcal{R}_3(C_0)$ in which p_{2k-i} writes to in $(\pi_{B_i}(C_0); \alpha_i)$. Since $\text{sig}(C_0) = \text{sig}(C_1)$, we have $r \notin \mathcal{R}_3(C_1)$, and r is covered by at most two processes in C_0 as well as in C_1.

Let τ be the shortest prefix of α_i such that p_{2k-i} is about to write to r in $\pi_{B_i}\tau(C_0)$. Since p_{2k-i} does not participate in schedule $\pi_{B_{1-i}}\pi_{B_2}\eta$, it is also covering r in the configuration $\pi_{B_i}\tau\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)$. Note that configurations $\pi_{B_i}\pi_{B_{1-i}}\pi_{B_2}(C_0)$ and $\pi_{B_{1-i}}\pi_{B_i}\pi_{B_2}(C_0)$ are indistinguishable to all processes. Therefore, $\pi_{B_i}\pi_{B_{1-i}}\pi_{B_2}\eta(C_0) = C_1$. Moreover, since $C_1 = \pi_{B_0}\pi_{B_1}\pi_{B_2}\eta(C_0)$ is indistinguishable from $\pi_{B_i}\tau\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)$ to every process except p_{2k-i}, all processes other than p_{2k-i} cover the same registers in C_1 as in $\pi_{B_i}\tau\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)$. Since p_{2k-i} covers r in this configuration, and r is covered by at most 2 other processes, $\pi_{B_i}\tau\pi_{B_{1-i}}\pi_{B_j}\eta(C_0)$ is a $(3,k)$-configuration.

□

Finally we state the main theorem of this chapter.

Theorem 1. *Any long-lived unbounded timestamp implementation that satisfies non-deterministic solo termination uses at least $n/6 - O(1)$ registers.*

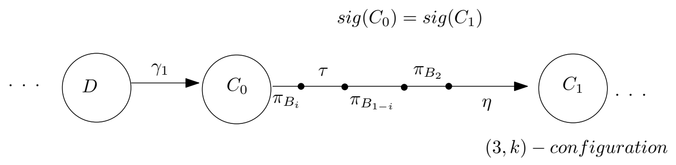


Figure 3.3: Illustration of Lemma 3

Proof. Lemma 3 shows that in any long-lived unbounded timestamp implementation that satisfies non-deterministic solo termination there exists a reachable $(3, \lceil (n-1)/2 \rceil)$-configuration. Clearly, in this configuration at least $\lceil (n-1)/6 \rceil = n/6 - O(1)$ registers are covered. □

Our first result closes a big gap between the lower and upper lower bounds of bounded timestamp implementations, stating that linear register space is necessary to implement a long-lived bounded timestamp system that satisfies non-deterministic solo termination (bounded timestamps system that uses $O(n)$ registers are well known). The next chapter presents a lower bound on the necessary number of registers for any one-shot timestamp implementation.

Chapter 4

Space Bounds For One-shot Timestamps

4.1 The One-Shot Timestamp Problem

The proofs we present in this chapter are part of the second principal result we obtained in a research project regarding bounded timestamps at the University of Calgary under the supervision of Dr. Lisa Higham and Dr. Philipp Woelfel. Original proofs and algorithms can be found in [28].

In Chapter 1, we mentioned that several objects have simpler implementations if we consider their one-shot versions. One example is *renaming* objects [5]. Mark Moir and James Anderson proved that in order to implement a long-lived renaming object, complex resettable *building blocks* are necessary [31], while a one-shot renaming object requires simpler auxiliary objects. They also proved that only using reads and writes, a one-shot $k(k+1)/2$-renaming object can be implemented with time complexity $O(k)$ while the resulting time complexity to implement the long-lived counterpart is $O(nk)$, where k is the number of processes actually requiring a name. They also prove that the one-shot solution can be combined with other previous results to obtain a $(2k-1)$-renaming solution. It remains as an open question whether or not there exists a $(2k-1)$-renaming solution for the long-lived version of the problem.

During the last decade, there have been an increasing use of algebraic topology to study distributed systems. Lower bounds for one-shot object implementations have been obtained using this tool [15]. Also there are other problems in distributed computing that are inherently one-shot as consensus or non-resettable test and set objects [28]. Hence, we consider that studying one-shot objects is of great interest. Specially, one-shot timestamp objects which as far we are aware have never been studied before.

```
Function getTS
  // In $p_{me}$ $me$ is the id of $p$, where $me \in \{1, \dots, n\}$
  // $R[1 \dots \lceil n/2 \rceil]$ is a shared array of multi-reader/2-writer registers each with a value
     in $\{0, 1, 2\}$ and initialized to 0. Register $R[i]$ is written by processes $p_{2i-1}$ and $p_{2i}$.
  // sum is a local variable
  sum := 0
  for $i = 1 \dots \lceil n/2 \rceil$ do
      if $i = \lceil me/2 \rceil$ then $R[i] := R[i] + 1$
      sum := sum + $R[i]$
  end
  return sum

Function compare($t_1, t_2$)
  return $t_1 < t_2$
```

Figure 4.1: The One-Shot Timestamp Algorithm.

4.2 A One-Shot Timestamp Algorithm

Results in Chapter 3 establish that linear space is needed to implement any long-lived timestamp mechanism. So it seems natural to imagine that n registers would be always required to construct a timestamp mechanism for a system of n processes. However, this is not the case if we consider a slightly different version of the problem. In this chapter we study the space complexity of one-shot unbounded timestamp implementations. Recall that a timestamp implementation is one-shot if processes are allowed to get a timestamp only once through invoking the `getTS` method. Processes can invoke the `compare` method many times. In this chapter, we present a wait-free algorithm that beats the best known algorithm for a long-lived implementation. The algorithm we present requires only $\lceil n/2 \rceil$ atomic `MWMR registers`. See Figure 4.1.

Each register is written to by only one pair of processes. Every process writes to only one register but it is allowed to read any of the $n/2$ shared registers. When a process p gets a timestamp, it first reads all the registers one by one adding up their values. When it encounters its register, before adding its value to the sum it first writes the read value plus one into it, then it keeps going until it reads all the registers. Then, it returns the sum as its timestamp. The `compare`(t_1, t_2) method simply uses the usual total order for integers $<$ to return the corresponding boolean value.

Lemma 4. *The Algorithm in Figure 4.1 is wait-free and implements a one-shot timestamp mechanism for an asynchronous system of n processes.*

Proof. Clearly both methods `getTS()` and `compare` are wait-free. Let p and q be two processors that perform a `getTS()` method call and let t_p and t_q be their corresponding timestamps. Assume that p's method call to `getTS()` happens before q's method call to `getTS()`. Each process writes either 1 or 2 to its register and only writes 2 if it observed that its register already held 1. Because it is one-shot, any such observed 1 must have been written by the observing process' partner, and thus the value in each register never decreases. Consequently, the value of `sum` also never decreases so $t_p \leq t_q$. Since p.`getTS()` happens before q.`getTS()`, q's `sum` will also account for the additional 1 that q writes to its own register and that is not observed by p. Therefore $t_p < t_q$. □

4.3 Space Lower Bound For One-Shot Timestamps

In this section we present a lower bound for one-shot timestamp implementations. We use a covering argument again but we use it in a different manner. In a one-shot timestamping system, processes are not allowed to perform more than one `getTS`, so the repetition of signatures through the repeatedly invocation of the `getTS` method is no longer possible. In the next subsection we give a geometric representation of covering structures that inspired the proof in section 4.3.2.

4.3.1 Geometric Intuition

In a one-shot timestamp system, processes are allowed to get a timestamp only once. To do so, most of them write to some register. Given a one-shot timestamp configuration C, we may give a *picture* of C in which we describe the writes to registers that have been performed and the writes that are about to be performed. Assume there are exactly $\sqrt{n}$ `MWMR registers` (assume that $\sqrt{n}$ is integer), where n is the total number of processes in the system. We represent some configurations a one-shot timestamping mechanism passes through by using a grid of $\sqrt{n} \cdot \sqrt{n}$ cells, see Figure 4.2. We assign to each register in the system one column of such a grid, such that each register is the *owner* of exactly $\sqrt{n}$ cells. The cells of the grid can be *filled* with tokens of different colors. If a process is covering some register r in C (a write to r is about to take place) we assign a green token to that process and we set it into the *lowest* empty cell on r's column. When that process writes to register r we replace its green token with a black one. In any cell there is always only one token. Therefore, writes that are about to be performed are represented by green tokens and writes that already happened are represented by black ones. As soon as a token is assigned to some process, that process is considered unavailable, then in any one-shot timestamp configuration C there are as many available processes as the number of empty cells in the picture of C. Figure 4.2 depicts some configuration C of a one-shot timestamp system in which one write has been performed on registers r_1 and r_2. This picture also tells us that registers r_1, r_2 and r_3 are covered by three processes, two processes and one process respectively. The number of available processes in C is $n - 8$. It is allowed to rearrange tokens on a column. For example, in Figure 4.2 we can move A to B's place and vice versa. For convenience, we assume that there are not empty cells between filled cells of a column.

Given a configuration C, we define the *cover size* of r as the number of processes covering r (the number of green tokens on r's column in the picture of C). In the picture of any configuration C, registers always appear arranged in order of decreasing cover size. To do so, we rename the register whose green column is the *highest* as r_1 and so on, such that the cover size of r_i is bigger than or equal to the cover size of r_k for all $k \geq i$, see Figure 4.3.

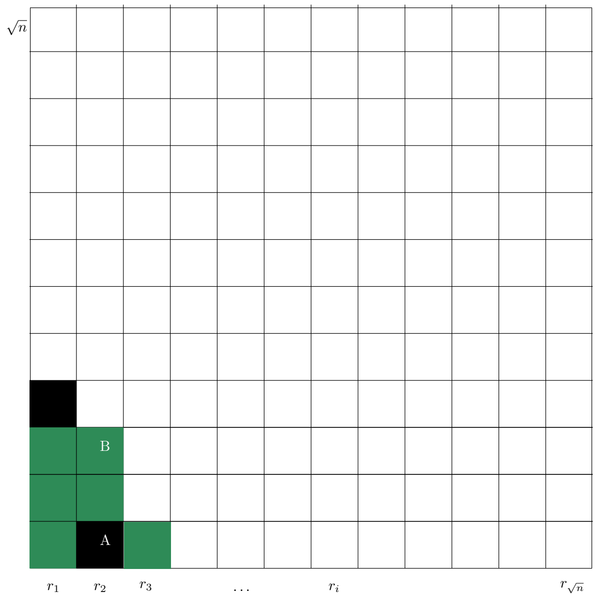


Figure 4.2: Picture of C.

Consider the initial configuration C^* in which all processes of the system are available. Let p be a process whose $\{p\}$-only execution (C^*, σ_1) contains a complete `getTS()` instance and writes to some register r. Let us consider the execution $(C^*, \widehat{\sigma_1})$ in which p is about to write to r. Then, in the picture of $\widehat{\sigma_1}(C^*)$ there is only one green token on r's column. We repeat this but now starting from $\widehat{\sigma_1}(C^*)$. We keep doing this until the cover size of some r_i is equal to $\sqrt{n} - i + 1$. In other words, we *set* a green token on the registers' columns while they are not high enough to *touch* the diagonal FJ, see Figure 4.3. It is obvious that eventually the cover size of some register will touch FJ. This is because the number of possible green tokens that can be set on some column of the grid is as many as the empty cells in C^*. So we have plenty of green tokens to fill with them the white area below FJ. See Figure 4.3. Hence, we can assume that the procedure described above leads us to some configuration C_j in which there exists some register r_j whose cover size is equal to $\sqrt{n} - j + 1$ (recall that we rename registers according to their cover size), see Figure 4.3. Let $R_j = \{r_1, \ldots, r_j\}$. Note that in C_j each register in R_j is covered by at least $\sqrt{n} - j + 1$ processes. If $\sqrt{n} - j + 1 \geq 3$, then there are three disjoint sets of processes covering R_j, so we can apply Lemma 1 to find an execution in which after a block write on R_j, up to half of the available processes write outside of R_j. Assume that N contains those processes that write outside of R_j and let π be a schedule such that (C_j, π) is a block write on R_j. Then starting in $\pi(C_j)$ we can proceed as above to find an schedule σ, in which only processes in N participate, such that in $\pi\sigma(C_j)$ there exists a register r_m whose cover size is equal to $\sqrt{n} - m + 1$. Let $\pi\sigma(C_k) = C_m$. Using the token terminology, in the picture of C_m the column corresponding to r_m touches the diagonal. The argument is a geometric one and it goes as follows: Let W be the white space in the picture of C_j. Hence, W represents the number of available process in C_j. Also let S be the white space below the line TJ. Note that $S \leq \frac{W}{2}$. Therefore, after the block write (C_j, π) the number of processes that write outside of R_j suffices to fill the white area below the line TJ with green tokens. So using at most $|N|$ processes in σ we reach the desired configuration C_m. Figure 4.4 depicts configuration C_m. Black tokens represent those processes that participated in π, the block write on R_j, the light-green tokens represent processes that participated in σ. Finally note that in C_m each register in $R_m = \{r_1, \ldots, r_j \ldots r_m\}$ is covered by at least $\sqrt{n} - m + 1$ processes and $m > j$.

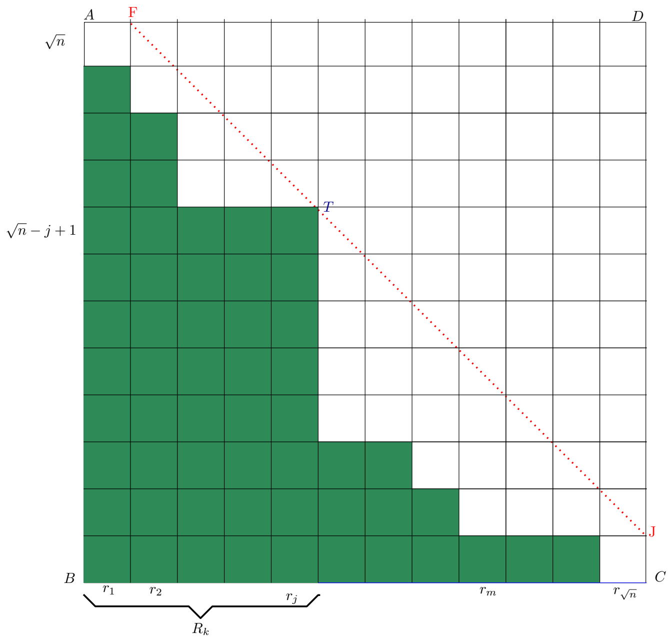


Figure 4.3: Picture of C_j.

4.3.2 A Lower Bound For One-Shot Timestamps

We now prove that any one-shot timestamp implementation that satisfies non-deterministic solo termination requires at least $\sqrt{n} - 1$ registers. The proof proceeds by inductively constructing a sequence of configurations $C_{\eta_0}, C_{\eta_1}, \ldots, C_{\eta_\ell}$ where the indices denoting these configurations are strictly increasing non-negative integers and each configuration is reachable from its predecessor. We will show that for each $m \in \{\eta_1, \ldots \eta_\ell\}$, in configuration C_m there is a set of m registers, R_m, such that each register in R_m is covered by at least $\sqrt{n} - m + 1$ processes. We call each of these configurations a *full configuration* and its corresponding set of registers the *full-covered* set of registers of that configuration. We will establish that $\eta_\ell \in \{\sqrt{n} - 1, \sqrt{n}\}$, thus implying that at least this many registers are covered in the last full configuration. To move from configuration C_{η_i} to $C_{\eta_{i+1}}$, we invoke Lemma 1. Let $\eta_i = j$. Lemma 1 guarantees that if the cover size of each register in R_j is at least three, there is a block-write to R_j, such that up to half of the available processes can be made to begin executing overlapping `getTS()` methods and manipulated to cover new registers outside of

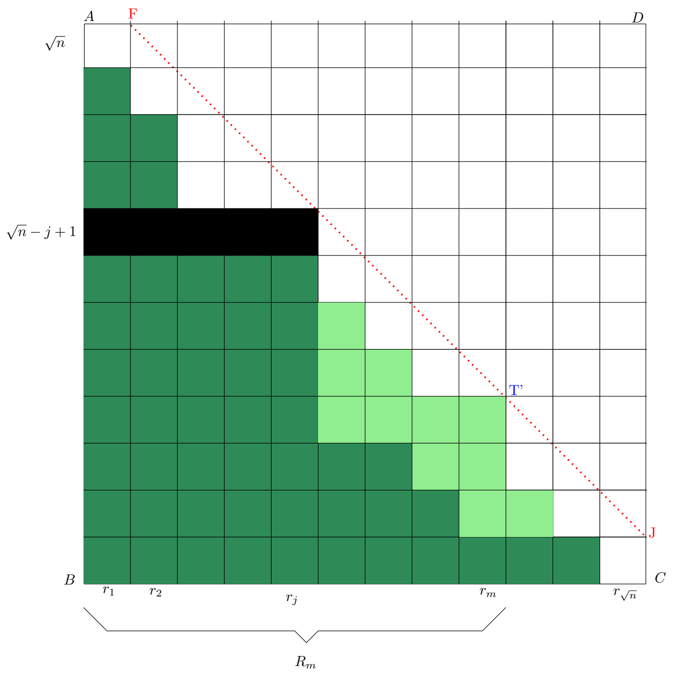


Figure 4.4: Picture of C_m.

R_j. We say a process p is *available* in some configuration C if it has not invoked the `getTS()` method. The core idea is to take one process of N and make it start a `getTS()` method but before it writes to some register outside of R_j we stop it, then we repeat the same procedure until we reach a new full configuration C_k whose set of full-covered registers R_k contains k registers, each of them covered by at least $\sqrt{n}-k+1$ processes and $k > j$. Therefore $C_k = C_{\eta_{i+1}}$. We show that it is always possible to reach such a configuration while there are enough available processes, the cover size of the full-covered registers is at least three, and we have not covered all the registers yet. This implies that in the last full configuration at least $\sqrt{n}-1$ registers are covered.

Algorithm 1 specifies the construction of a schedule ω that produces an execution passing through the full configurations described above. To achieve this, we define two auxiliary functions: $blockWrite(C,R,U)$ and $reachNewFull(C,R,\pi,N)$.

Suppose that R is the full-covered set of registers of some full configuration C, also assume that each register in R is covered by at least three processes. Let U denote the set of available processes in C. By Lemma 1, there exists a set of processes $N \subset U$ of size $\lfloor |U|/2 \rfloor$ and a block-write $(C;\pi)$ to R such that starting from $\pi(C)$, any N-only execution that contains a complete `getTS()` must write to some register outside of R. We define $blockWrite(C,R,U)$ as the function that returns such schedule π and set of processes N (If R is empty then π is the empty schedule ε).

Let $m = \lfloor |U|/2 \rfloor$ and $N = \{p_1, p_2, \ldots, p_m\}$ (recall that $N_k = \{p_1, \ldots, p_k\}$ for $1 \le i \le k$

and $N_0 = \emptyset$). We inductively construct schedule δ_i such that in $\pi\delta_i(C)$ all processes in N_i are about to write to some register not in R. Suppose δ_{i-1} is a N_{i-1}-only schedule such that in $\pi\delta_{i-1}(C)$ each process in N_{i-1} is about to write outside of R, $1 \leq i \leq m$. Let σ_i be a $\{p_i\}$-only schedule that contains a complete `getTS()` and $p_i \in N_i \setminus N_{i-1}$. Since execution $(C; \pi\delta_{i-1}\sigma_i)$ contains a complete `getTS()` instance, p_i writes to some register not in R. Let $\widehat{\sigma_i}$ be the shortest prefix of σ_i such that in $\pi\delta_{i-1}\widehat{\sigma_i}(C)$ process p_i is about to write outside of R. Therefore, in $\pi\delta_{i-1}\widehat{\sigma_i}(C)$ every process in N_i is about to write outside of R. Let $\delta_i = \delta_{i-1}\widehat{\sigma_i}$.

We define $reachNewFull(C, R, \pi, N)$ as the function that returns the pair $(\gamma, NewR)$ where $\gamma = \delta_i$ such that i is the smallest value in $\{1 \ldots, m\}$ such that $\pi\delta_i(C)$ is a full configuration whose set of full-covered registers $newR$ satisfies: $\forall r \in newR, |cover(r)| \geq \sqrt{n} - |newR| + 1$ and $|newR| > |R|$. If i does not exist , then $\gamma = \delta_i$ and $newR = R$.

In Algorithm 1, for any register r, $cover(r)$ denotes the cover size of r, $participate(\theta)$ denotes the set of processes whose indices appear in the schedule θ.

Algorithm 1: Constructing an execution using many registers

```
Output: A schedule $\omega$.
// $\varepsilon$ denotes the empty schedule.
// Initialization:
$U := \mathcal{P}$; $R := \emptyset$; $coverSize := 3$; $D := C^*$; $\omega := \varepsilon$
alg_construct()
while $|U| \geq 2 \land coverSize \geq 3 \land |R| < m$ do
    // Step 1: find $\pi$ and $N$ to cover registers not in $R$:
    $(\pi, N) := blockWrite(D, R, U)$
    // Step 2: find $\gamma$ to reach a new full configuration:
    $(\gamma, newR) := reachNewFull(D, R, \pi, N)$
    // Step 3: update
    $D := \pi\gamma(D)$
    $R := newR$
    $\omega := \omega\pi\gamma$
    $U := U \setminus participate(\gamma)$
    $coverSize := \sqrt{n} - |newR| + 1$
end
return $\omega$
```

Lemma 5. *In the execution constructed by Algorithm 1, no process initiates more than one* `getTS()` *method call.*

Proof. Let C^* be the initial configuration and let $(C^*; \omega)$ be the execution constructed by Algorithm 1. Every process that participates in ω after starting a `getTS()` method call either participates in a block write or covers some register. As long as a process is covering a register, it is paused during its first execution of `getTS()`. Any process that participated in a block-write at Step 1, takes no further steps. Thus, no process initiates `getTS()` more than once in $(C^*; \omega)$. □

Lemma 6. *Consider any implementation of timestamp objects that has a set of $m \leq \sqrt{n}$ registers, $\mathcal{R}$, available. The execution constructed by Algorithm 1 for this implementation satisfies the following: There exists a sequence of integers $\eta_0 < \eta_1 < \ldots < \eta_\ell$, where $\eta_\ell \geq m-1$, and reachable full configurations $C_{\eta_0}, C_{\eta_1}, \ldots, C_{\eta_\ell}$, such that each of these configurations is reachable from its predecessor.*

Proof. The proof proceeds inductively on the number of iterations of the while loop in Algorithm 1. Before entering to the loop, we have $\eta_0 = 0$, $\omega_0 = \epsilon$. So C_{η_0} is the initial configuration C^* in which no registers are covered, and all processes are in U, so the lemma holds for η_0.

Let ω_i be the schedule obtained at the end of iteration i. Assume that $\omega_i(C^*) = C_{\eta_i}$ is a reachable full configuration whose set of full-covered registers is R_{η_i}, and suppose we enter to the loop one more time. Let $j = \eta_i$. Then in $\pi(D)$ (Step 1 of the $i+1$-th iteration).

1. $j \cdot (\sqrt{n} - j)$ processes cover registers in R_j, and
2. the number of processes that have participated in any block-write at Step 1 is at most $(j(j+1)/2)$.

Since $|R_j| = j$, the total number of processes either already covering a register in $\mathcal{R} \setminus R_j$ or still in the set U in configuration $\pi(D)$ is at least $n - (\frac{2j \cdot \sqrt{n} - j^2 + j}{2})$. If at Step 2 of the $i+1$ iteration $reachNewFull(D, R, \pi, N)$ returns a γ such that $\pi\gamma(D)$ is a full configuration, clearly we are done. Suppose for the purpose of contradiction that $\pi\gamma(D)$ is not a full configuration satisfying the desired properties. Recalling the geometric intuition, this implies that after setting all processes in N to cover registers outside of R_j there is no a register whose cover size touches the line FJ, this assumption should lead us to a contradiction. In $\pi\gamma(D)$, each process in $particpate(\gamma)$ covers a register in $\mathcal{R} \setminus R_j$. Let $r_{j+1}, \ldots, r_m$ registers in $\mathcal{R} \setminus R_j$ arranged in order of decreasing cover size. Hence:

1. since $|N| = \lfloor |U|/2 \rfloor$ then $|participate(\gamma)| = \lfloor |U|/2 \rfloor$.
2. there are at least $\lfloor \frac{n - j \cdot \sqrt{n}}{2} + \frac{j^2 - j}{4} \rfloor$ processes covering registers in $\mathcal{R} \setminus R_j$.
3. for every z, $j+1 \leq z \leq m$, $|cover(r_z)| \leq \sqrt{n} - z$.

Thus the covering set on $\mathcal{R} \setminus R_j$ has size $S = \sum_{z=j+1}^{m} |cover(r_z)|$. Using $1 \leq j \leq \sqrt{n} - 1$, $2 \leq \sqrt{n}$ and $|R_j| = j$, we obtain

$$\begin{aligned} S \leq \sum_{z=j+1}^{\sqrt{n}} (\sqrt{n} - z) &= \sum_{z=0}^{\sqrt{n}-j-1} z \\ &= \frac{(\sqrt{n} - j - 1)(\sqrt{n} - j)}{2} \\ &< \frac{(\sqrt{n} - j - 1)\sqrt{n}}{2} \\ &= \frac{n - j \cdot \sqrt{n} - \sqrt{n}}{2} \\ &\leq \left\lfloor \frac{n - j \cdot \sqrt{n}}{2} \right\rfloor \end{aligned}$$

This contradicts that at least $\lfloor \frac{n - j \cdot \sqrt{n}}{2} + \frac{j^2 - j}{4} \rfloor$ processes cover registers in $\mathcal{R} \setminus R_j$ in $\pi\gamma(D)$. Such a contradiction comes from assuming $\pi\gamma(D)$ is not a full configuration that holds the required properties. □

Theorem 2. *Any implementation of a one-shot timestamp system that satisfies non-deterministic solo termination uses at least $\sqrt{n} - O(1)$ registers.*

Proof. It follows from the two previous lemmas. □

Chapter 5

Future Research and Conclusions

The consensus number of an object of type O is the largest m for which that object together with atomic read/write registers solves the m-process consensus problem. See [9] for further information about the consensus problem. Then we say an object A is *stronger* than object B if A's consensus number is bigger than B's. The type of registers we considered throughout this work were only read/write atomic registers whose consensus number is one [22]. On the other hand, recall that we mentioned in Chapter 2 that there were other types of `register` that support more powerful operations such as `CAS`, `TEST&SET` or `Read-Write-Modify functions`. These registers have higher consensus number. Thus, a natural question emerges: can a timestamp object be implemented with some strong objects more efficiently in time or space than implementations that use only read/write atomic registers?. Studying the relationship between timestamping objects and other distributed objects can be of interest to understand how difficult the timestamping problem is.

In this last chapter we discuss all these issues and we propose some questions we consider interesting for future research.

5.1 Strong Objects

The `CAS` operation takes two arguments `old` and `new` such that if `x` is a register that supports `CAS` operations, then `x.CAS(old,new)` returns a boolean value. If the value of `x` is equal to `old`, it is atomically replaced by `new` and `true` is returned, otherwise the value of `x` is unchanged and `false` is returned. Notice that the consensus number of a `CAS` register is infinite. It is easy to see that wait-free timestamping systems are constructible from `CAS` registers. Figure 5.1 shows the construction of an unbounded long-lived timestamping system using only one `CAS` register.

However, sometimes the cost of including a `CAS` register in a system may be too high, so it is important to investigate if timestamp systems are constructible from weaker and cheaper objects. Giving lower bounds on the number of stronger objects needed to implement a timestamp system and to show a timestamp system implementation from such objects is an interesting direction for future research.

```
**Function** getTS
    $last := x.read()$
    $x.CAS(last, last + 1)$
    **return** $last$
```

```
**Function** compare($t_1, t_2$)
    **return** $t_1 < t_2$
```

Figure 5.1: CAS timestamping

5.2 Lattice Agreement

The lattice agreement problem for eponymous systems is a decision problem introduced by Hagit Attiya, Maurice Herlihy and Ophir Rachman [6]. In the shared memory model a process is said to be active in some execution $(C; \alpha)$ if its index appears in α. Assume that there exists a bound on the number of active process in any execution and this bound is known in advance. The lattice agreement problem is defined as follows: *A process p_i starts with $V_{in}(p_i) = \{p_i\}$ and is required to decide on a subset of the active processes, called a view, $V_{out}(p_i)$, such that the following conditions hold:*

- Comparability: for any i and j, either $V_{out}(p_i) \subseteq V_{out}(p_j)$ or $V_{out}(p_j) \subseteq V_{out}(p_i)$.
- Self-containment: for any i, $V_{in}(p_i) \subseteq V_{out}(p_i)$.

It seems that there exists a relationship between the lattice agreement problem and the one-shot timestamp problem. In order to implement the `getTS` method from the outcome of a lattice agreement object, we have to ensure that if the `getTS` method of process p_i happens before the `getTS` method of process p_j, then the timestamp constructed from $V_{out}(p_i)$ must be smaller than the one granted to p_j (constructed from $V_{out}(p_j)$). If such construction exists we could solve the timestamp problem through lattice agreement objects.

There are many other questions that remain open regarding timestamps. For instance, regarding the timestamp universe, we can study the necessary number of registers to implement a one-shot timestamp if the timestamp universe is nowhere dense. Also it remains a big gap between the lower bound and the upper bound for one-shot timestamping implementations that it seems possible to shrink. Also it remains open to find an interesting application where one-shot timestamps may be used.

5.3 Conclusions

Throughout this thesis we studied the space complexity of timestamp implementations. We considered an asynchronous system with n processes that communicate among themselves through performing operations on m read/write atomic shared registers. We first described two types of long-lived timestamp systems: those that provide timestamps whose size is not bounded and those that provide timestamps whose size is bounded. Timestamps in a bounded timestamp mechanism are elements of a finite timestamp universe, $\mathcal{T}$, while in a unbounded one, they are elements of an infinite timestamp universe. We shortly described a bounded timestamp implementations due to Cinthya Dwork and Orli Waarts [13], but our results hold for unbounded timestamp systems only. Faith Ellen, Panagiota Fatourou and Eric Ruppert [14] proved that $\Omega(\sqrt{n})$ `MWMR registers` are needed to implement a long-lived unbounded timestamp system that satisfies non-deterministic solo termination. As far we are aware, such lower bound is the only known result regarding the space complexity of unbounded timestamp implementations. We presented an improvement to that lower bound in Chapter 3. We did so, stating that $n/6 - O(1)$ `MWMR registers` are required for implementing a timestamp mechanism that satisfies non-deterministic solo termination. This result closes the large gap between the upper bound of $O(n)$ also proved in [14] and the $\Omega(\sqrt{n})$ lower bound. Our proof use a covering argument, which is a technique introduced some years ago by James Burns and Nancy Lynch [10]. We show how to use that technique together with the idea that in a timestamp system we can repeat certain configurations many times without using many processes. After setting up the appropriate induction hypothesis proving the new lower bound is straightforward. A one-shot timestamp mechanism that satisfies wait-freedom is presented in Chapter 4. Such an algorithm uses only $n/2$ `MWMR registers` while the best long-lived implementation of a timestamp system uses $n-1$ `SWMR` registers. Furthermore, we prove a lower bound for one-shot timestamp implementations. We proved that $\sqrt{n} - O(1)$ `MRMW registers` are necessary to implement any one-shot timestamp system that satisfies non-deterministic solo termination. Lower bound proof in Chapter 4 was inspired by a geometric interpretation of the covering structure of configurations. Chapter 4 results leave a large gap between the upper bound and lower bound of one-shot timestamp implementations. Our intuition is that we can improve such a lower bound although it remains as an open question. We presented some open questions regarding the space complexity of timestamp implementations using stronger objects. We gave an implementation that uses only one CAS register, although it remains open to state lower bounds and algorithms using other objects. The last section discussed the possible relationship of timestamping to the lattice agreement problem. Timestamp systems are a quiet useful tool in distributed computing and provides many interesting research problems.

We hope you had fun while you read this work. All results presented were obtained during my research stay at the University of Calgary under the supervision of Dr. Lisa Higham and Dr. Philipp Woelfel.

Bibliography

[1] Karl Abrahamson. On achieving consensus using a shared memory. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 291–302, New York, NY, USA, 1988. ACM.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[3] Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, 1994.

[4] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441–461, 1990.

[5] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.

[6] Hagit Attiya and Arie Fouren. Adaptive wait-free algorithms for lattice agreement and renaming (extended abstract). In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 277–286, New York, NY, USA, 1998. ACM.

[7] Hagit Attiya and Arie Fouren. Adaptive wait-free algorithms for lattice agreement and renaming (extended abstract). In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 277–286, 1998.

[8] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, 2003.

[9] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[10] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[11] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.

[12] Cynthia Dwork, Maurice Herlihy, Serge A. Plotkin, and Orli Waarts. Time-lapse snapshots. *SIAM Journal On Computing*, 28(5):1848–1874, 1999.

[13] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. ACM*, 46(5):633–666, 1999.

[14] Faith Ellen, Panagiota Fatourou, and Eric Ruppert. The space complexity of unbounded timestamps. *DISC'07:Proceedings of the twenty first International Symposium on Distributed Computing*, 21(2):103–115, 2008.

[15] Faith Ellen and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

[16] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, 1988.

[17] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, 1989.

[18] Rainer Gawlick, Nancy A. Lynch, and Nir Shavit. Concurrent timestamping made simple. In *1st Israel Symposium on Theory of Computing Systems (ISTCS)*, pages 171–183, 1992.

[19] Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[20] Sibsankar Haldar and Paul M. B. Vitányi. Bounded concurrent timestamp systems using vector clocks. *J. ACM*, 49(1):101–126, 2002.

[21] Danny Hendler and Philipp Woelfel. Randomized mutual exclusion in $o(\log n/\log\log n)$ rmrs. In *PODC 09: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 26–35, 2009.

[22] Maurice Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[23] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[24] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[25] Amos Israeli and Meir Pinhasov. A concurrent time-stamp scheme which is linear in time and space. In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 95–109, London, UK, 1992. Springer-Verlag.

[26] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[27] Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(4):723–746, 1996.

[28] Higham Lisa, Pacheco Eduardo, and Woelfel Philipp. Lower bounds for timestamp implementations (draft).

[29] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 147–172, London, UK, 1979. Springer-Verlag.

[30] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[31] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Computer Science Program.*, 25(1):1–39, 1995.

[32] Susan Speer Owicki. *Axiomatic proof techniques for parallel programs.* PhD thesis, Ithaca, NY, USA, 1975.

[33] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[34] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 233–243, 1986.