Revisionist Simulations: A New Approach to Proving Space Lower Bounds

(Extended Abstract)

Faith Ellen
University of Toronto
faith@cs.toronto.edu

Rati Gelashvili
University of Toronto
gelash@cs.toronto.edu

Leqi Zhu
University of Toronto
lezhu@cs.toronto.edu

ABSTRACT

Determining the number of registers required for solving x -obstruction-free (or randomized wait-free) k -set agreement for $x \leq k$ is an open problem that highlights important gaps in our understanding of the space complexity of synchronization. In x -obstruction-free protocols, processes are required to return in executions where at most x processes take steps. The best known upper bound on the number of registers needed to solve this problem among $n > k$ processes is $n - k + x$ registers. No general lower bound better than 2 was known.

We prove that any x -obstruction-free protocol solving k -set agreement among $n > k$ processes must use $\lfloor \frac{n-x}{k+1-x} \rfloor + 1$ or more registers. Our main tool is a simulation that serves as a reduction from the impossibility of deterministic wait-free k -set agreement. In particular, we show that, if a protocol uses fewer registers, then it is possible for $k + 1$ processes to simulate the protocol and deterministically solve k -set agreement in a wait-free manner, which is impossible. An important aspect of the simulation is the ability of simulating processes to revise the past of simulated processes. We introduce an *augmented snapshot object*, which facilitates this.

We also prove that any lower bound on the number of registers used by obstruction-free protocols applies to protocols that satisfy nondeterministic solo termination. Hence, our lower bound of $\lfloor \frac{n-1}{k} \rfloor + 1$ for the obstruction-free case (i.e., $x = 1$) also holds for randomized wait-free protocols. In particular, we get a tight lower bound of exactly n registers for solving obstruction-free and randomized wait-free consensus.

KEYWORDS

set agreement; space complexity; lower bounds; obstruction-freedom

ACM Reference Format:

Faith Ellen, Rati Gelashvili, and Leqi Zhu. 2018. Revisionist Simulations: A New Approach to Proving Space Lower Bounds : (Extended Abstract). In *PODC '18: ACM Symposium on Principles of Distributed Computing, July 23-27, 2018, Egham, United Kingdom*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3212734.3212749>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions.acm.org.

PODC '18, July 23-27, 2018, Egham, United Kingdom

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5795-1/18/07...\$15.00

<https://doi.org/10.1145/3212734.3212749>

He who controls the past controls the future. He who controls the present controls the past.

George Orwell, 1984

1 INTRODUCTION

The k -set agreement problem is a classical synchronization task, introduced by Chaudhuri [19], in which n processes, each with an input value, are required to return at most k different input values. This is a generalization of the consensus problem, which is the case $k = 1$.

Two celebrated results in distributed computing are the impossibility of solving consensus deterministically when at most one process may crash [25, 37] and, more generally, the impossibility of solving k -set agreement deterministically when at most k processes may crash [14, 34, 40], using only registers. One way to bypass these impossibility results is to design protocols that are obstruction-free [33]. *Obstruction-freedom* is a termination condition that requires a process to terminate given sufficiently many consecutive steps, i.e., from any configuration, if only one process takes steps, then it will eventually terminate. *x -obstruction-freedom* [43] generalizes this condition: from any configuration, if only x processes take steps, then they will all eventually terminate. It is known that k -set agreement can be solved using only registers in an x -obstruction-free way for $1 \leq x \leq k$ [44]. Another way to overcome the impossibility of solving consensus is to use randomized wait-free protocols, where non-faulty processes are required to terminate with probability 1 [13].

It is possible to solve consensus for n processes using n registers in a randomized wait-free way [1, 3, 5, 39] or in an obstruction-free way [16, 17, 30, 45]. A lower bound of $\Omega(\sqrt{n})$ was proved by Ellen, Herlihy, and Shavit in [23]. Recently, Gelashvili proved an $\Omega(n)$ lower bound for *anonymous* processes [28]. Anonymous processes [7, 23] have no identifiers and run the same code: all processes with the same input start in the same initial state and behave identically until they read different values. Then Zhu proved that any obstruction-free protocol solving consensus for n processes requires at least $n - 1$ registers [46]. All these lower bounds are actually for protocols that satisfy *nondeterministic solo termination* [23], which includes both obstruction-free and randomized wait-free protocols.

In contrast, there are big gaps between the best known upper and lower bounds on the number of registers needed for k -set agreement. The best obstruction-free protocols require $n - k + 1$ registers [16, 45]. Bouzid, Raynal, and Sutra [16] also give an x -obstruction-free protocol that uses $n - k + x$ registers, improving on the $\min(n + 2x - k, n)$ space complexity of Delporte-Gallet, Fauconier, Gafni, and Rajsbaum's obstruction-free protocol [20]. All of

these algorithms work for anonymous processes. Delporte-Gallet, Fauconnier, Kuznetsov, and Ruppert [21] proved that it is impossible to solve k -set agreement using 1 register. For anonymous processes, they also proved a lower bound of $\sqrt{x(\frac{n}{k} - 2)}$ for x -obstruction-free protocols, which still leaves a polynomial gap between the lower and upper bounds.

There are good reasons why proving lower bounds on the number of registers needed for k -set agreement may be difficult. At a high level, the impossibility results for k -set agreement consider some representation (for example, a simplicial complex) of all possible process states in all possible executions. Then, a combinatorial property (Sperner's Lemma [42]) is used to prove that, roughly speaking, for any given number of steps, there exists an execution leading to a configuration in which $k + 1$ outputs are still possible. Although there is ongoing work to develop a more general theory [26, 27, 41], we do not know enough about the topological representation of protocols that are x -obstruction-free or use fewer than n multi-writer registers [32] to adapt topological arguments to prove space lower bounds for k -set agreement. There are similar problems adapting known proofs that do not explicitly use topology [4, 10].

Our contribution. In this paper, we prove a lower bound of $\lfloor \frac{n-x}{k+1-x} \rfloor + 1$ on the number of registers necessary for solving n -process x -obstruction-free k -set agreement. As corollaries, we get a tight lower bound of n registers for obstruction-free consensus and a tight lower bound of 2 for obstruction-free $(n - 1)$ -set consensus. In the full version of the paper, we also prove a lower bound of $\lfloor \frac{n}{2} \rfloor + 1$ registers for obstruction-free ϵ -approximate agreement [22], for sufficiently small $\epsilon > 0$. This is another important task for which no good space lower bound was known.

In addition, in Section 5, we prove that any lower bound on the number registers needed for obstruction-free protocols to solve a task also applies to nondeterministic solo terminating protocols and, in particular, to randomized wait-free protocols solving that task. Hence, our space lower bounds for obstruction-free protocols also apply to such protocols.

Technical Overview. Using a novel simulation, we convert any obstruction-free protocol for k -set agreement that uses too few registers to a protocol that solves wait-free k -set agreement using only registers. Since solving wait-free k -set agreement is impossible using only registers, this reduction gives a lower bound on the number of registers needed to solve obstruction-free k -set agreement. This simulation technique, described in detail in Section 4, is the main technical contribution of the paper. It is the first technique that proves lower bounds on space complexity by applying results obtained by topological arguments. In the full version of the paper, we also use this new technique to prove a lower bound on the number of registers needed for ϵ -approximate agreement by a reduction from a step complexity lower bound for ϵ -approximate agreement.

The executions of the simulated processes in our simulation are reminiscent of the executions constructed by adversaries in covering arguments [6, 18]. In those proofs, the adversary modifies an execution it has constructed by revising the past of some process, so that the old and new executions are indistinguishable to the other processes. It does so by inserting consecutive steps of the process starting from some carefully chosen configuration. In our

simulation, a real process may revise the past of a simulated process, in a way that is indistinguishable to other simulated processes. This is possible because each simulated process is simulated by a single real process. In contrast, in the BG simulation [15], different steps of simulated processes can be performed by different real processes, so this would be much more difficult to do.

A crucial component of our simulation is the use of an *augmented snapshot object*, which we implement in a non-blocking manner from registers. Like a standard snapshot object, this object consists of a fixed number of components and supports a Scan operation, which returns the contents of all components. However, it generalizes the update operation to a Block-Update operation, which can update multiple components of the object. In addition, a Block-Update returns some information, which is used by our simulation. The specifications of Block-Update and our implementation of an augmented snapshot object appears in Section 3.

2 PRELIMINARIES

An asynchronous shared memory system consists of a set of processes and instances of base objects, which processes use to communicate. An *object* has a set of possible values and a set of operations, each of which takes some fixed number of inputs and returns a response. The processes take steps at arbitrary speeds and may fail, at any time, by crashing. Every step consists of an operation on some base object by some process plus local computation by that process to determine its next state from the response returned by the operation.

Configurations and Executions. A *configuration* of a system consists of the state of each process and the value of each object. An *initial* configuration is determined by the input value of each process. Each object has the same value in all initial configurations. A configuration C is *indistinguishable* from a configuration C' to a set of processes $\mathcal{P}$ in the system, if every process in $\mathcal{P}$ is in the same state in C as it is in C' and each object in the system has the same value in C as in C' .

A step e by a process p is *applicable* at a configuration C if e can be the next step of process p given its state in C . If e is applicable at C , then we use Ce to denote the configuration resulting from p taking step e at C . A sequence of steps $\alpha = e_1, e_2, \dots$ is *applicable* at a configuration C if e_1 is applicable at C and, for each $i \geq 1$, e_{i+1} is applicable at $Ce_1 \dots e_i$. In this case, α is called an *execution from C* . An execution $\alpha\beta$ denotes the execution α followed by the execution β . A configuration C is *reachable* if there exists a finite execution from an initial configuration that results in C .

For a finite execution α from a configuration C , we use $C\alpha$ to denote the configuration reached after applying α to C . If α is empty, then $C\alpha = C$. We say an execution α is *$\mathcal{P}$ -only*, for a set of processes $\mathcal{P}$, if all steps in α are by processes in $\mathcal{P}$. A $\{p\}$ -only execution, for some process p , is also called a *solo execution by p* . Note, if configurations C and C' are indistinguishable to a set of processes $\mathcal{P}$, then any $\mathcal{P}$ -only execution from C is applicable at C' .

Implementations and Linearizability. An *implementation* of an object specifies, for each process and each operation of the object, a deterministic procedure describing how the process carries out the operation. The *execution interval* of an invocation of an operation in an execution is the subsequence of the execution that

begins with its first step and ends with its last step. If an operation does not complete, for example, if the process that invoked it crashed before receiving a response, then its execution interval is infinite. An implementation of an object is *linearizable* if, for every execution, there is a point in each operation's execution interval, called the *linearization point* of the operation, such that the operation can be said to have taken place atomically at that point [35]. This is equivalent to saying that the operations can be ordered (and all incomplete operations can be given responses) so that any operation which ends before another one begins is ordered earlier and the responses of the operations are consistent with the sequential specifications of the object [35].

Progress Conditions. An implementation of an object is *wait-free* if every process is able to complete its current operation on the object after taking sufficiently many steps, regardless of what other processes are doing. An implementation is *non-blocking* if infinitely many operations are completed in every infinite execution.

A protocol is *x-obstruction-free* if, from any configuration C and for any subset P of at most x processes, every process in P that takes sufficiently many steps after C outputs a value, as long as only processes in P take steps after C . A protocol is *obstruction-free* if it is 1-obstruction-free and *wait-free* if it is n -obstruction-free.

Registers and Snapshot objects. A *register* is an object that supports two operations, write and read. A $\text{write}(v)$ operation writes value v to the register, and a read operation returns the last value that was written to the before the read. A *multi-writer register* allows all processes to write to it, while a *single-writer register* can only be written to by one fixed process. A process is said to be *covering* a register if its next step is a write to this register. A *block write* is a consecutive sequence of write operations to different registers performed by different processes.

An *m-component multi-writer snapshot* object [2] stores a sequence of m values and supports two operations, update and scan. An $\text{update}(j, v)$ operation sets component j of the object to v . A scan operation returns the current *view*, consisting of the values of all components. A *single-writer snapshot* object shared by a set of processes has one component for each process and each process may only update its own component. A process is said to be *covering* component j of a snapshot object if its next step is an update to the component j . A *block update* is a consecutive sequence of update operations to different components of a snapshot object performed by different processes.

It is easy to implement m registers from an m -component multi-writer snapshot object, by replacing each write to the j 'th register by an update to the j 'th component and replacing a read to the j 'th register by a scan and then discarding all but the value of the j 'th component. An m -component snapshot object can also be implemented from m registers [2].

Tasks and Protocols. A *task* specifies a set of allowable combinations of inputs to the processes and, for each such combination, what combinations of outputs can be returned by the processes. A *protocol* for a task provides a procedure for each process to compute its output, so that the task's specifications are satisfied. In particular, in *k-set agreement*, each process begins with an arbitrary value as its input and, if it does not crash, must output a value such that at most k values are output (*k-agreement*) and each output value is the input of some process (*validity*).

The *space complexity* of a protocol is the maximum number of registers used in any execution of the protocol. Each m -component snapshot object it uses counts as m registers. The *space complexity* of a task is the minimum space complexity of any protocol for the task.

2.1 Our Setting

We consider two asynchronous shared memory systems, the simulated system and the real system.

Simulated system. The *simulated system* consists of n simulated processes, $p_1, \dots, p_n$, that communicate through an m -component multi-writer snapshot object. Thus, any task that can be solved in the simulated system has space complexity at most m .

Without loss of generality, we assume that each process p_i alternately performs scan and update operations on the snapshot object: Between two consecutive update operations, p_i can perform a scan and ignore its result. If p_i is supposed to perform multiple consecutive scans, it can, instead, perform one scan and use its result as the result of the others. This is because it is possible for all these scans to occur consecutively in an execution, in which case, they *would* all get the same result.

Real system. The real system consists of $k + 1$ real processes, $q_0, \dots, q_k$, that communicate through a single-writer snapshot object. For clarity of presentation, real processes use single-writer registers in addition to the single-writer snapshot object. The single-writer registers to which a particular process writes can be treated as additional separate fields of the component of the snapshot object belonging to that process. In Section 3, we define and implement an *m-component augmented snapshot object* shared by the real processes.

In our simulation, the processes in the simulated system are partitioned into $k + 1$ sets, $P_0, \dots, P_k$, and real process q_i is solely responsible for simulating the actions of all processes in P_i in the simulated system. This is illustrated in Figure 1.

The following impossibility result applies to our real system.

THEOREM 1 ([14, 34, 40]). *In a system of $k + 1$ processes sharing a single-writer snapshot object, there is no deterministic wait-free protocol that allows these processes to solve k -set agreement.*

3 AUGMENTED SNAPSHOT OBJECT

In this section, we define an augmented snapshot object and show how it can be deterministically implemented in the real system. This object plays a central role in our simulation. It is used by real processes to simulate steps performed by simulated processes. In particular, a real process q_i uses this object to simulate an update or a scan by any simulated process in P_i , or a block update by any subset of processes in P_i . Our simulation, which is explained in Section 4, is non-standard. Unfortunately, to satisfy its technical requirements, the augmented snapshot has to satisfy some non-standard properties.

An m -component augmented snapshot object is a generalization of an m -component multi-writer snapshot object. A Scan operation returns the current *view*, consisting of the values of all components. The components can be updated using a Block-Update operation. The key difference between a multi-writer snapshot and

Figure 1: High-level overview of real and simulated systems.

an augmented snapshot is that a Block-Update may update *multiple* components, although not necessarily atomically. In addition, a Block-Update may return a view from some earlier point in the execution. Otherwise, it returns a special *yield* symbol, ∇ .

A linearizable, non-blocking implementation of an augmented snapshot object in the real system is impossible. This is because a Block-Update operation that updates 2 components would then be the same as a 2-assignment operation. However, 2-assignment, together with read or Scan, can be used to deterministically solve wait-free consensus among 2 processes [31], which is impossible in the real system [2, 37].

Instead, a Block-Update operation can be considered to be a sequence of atomic Update operations, which each update one component of the augmented snapshot object. (Analogously, a *collect* operation [9, 12] is not atomic, but the individual reads that comprise it are atomic.)

3.1 Specification

An *m*-component augmented snapshot object, $\mathcal{M}[1..m]$, shared by $k+1$ processes, $q_0, q_1, \dots, q_k$, consists of m components and supports two operations, Scan and Block-Update, which can be performed by all processes. A *view* of the augmented snapshot consists of the value of each of the m components at some point in an execution. A Scan operation returns the current view. A Block-Update operation to a sequence of $r \geq 1$ different components $[j_1, \dots, j_r]$ of $\mathcal{M}$ with a sequence of r values $[v_1, \dots, v_r]$ is comprised of a sequence of Update operations $U_1, \dots, U_r$. Each Update U_c atomically sets $\mathcal{M}[j_c]$ to v_c . These Update operations may occur in any order. A Block-Update operation also returns either ∇ or a view of $\mathcal{M}$.

A Block-Update that does not return ∇ is called *atomic*. We require that every execution has a linearization in which the Update operations comprising each atomic Block-Update are linearized consecutively.

Consider any atomic Block-Update, B . Let Z be the first Update in B . Let Z' be the last Update prior to Z that is part of an atomic Block-Update or the beginning of the execution, if there is no such Update. Then B must return a view of $\mathcal{M}$ at some point T between Z' and Z such that no Scan occurs between T and Z .

We prove that, in our implementation, a Block-Update only returns ∇ under certain circumstances, as described in Theorem 3. For example, a Block-Update by process q_0 will always be atomic and, if a Block-Update experiences no step contention [8], it will be atomic. The simulation in Section 4 relies on this property of our implementation.

3.2 Implementation

Figure 2 contains the pseudocode for our non-blocking implementation of an m -component augmented snapshot object, $\mathcal{M}$, shared by $k+1$ processes, $q_0, q_1, \dots, q_k$. Every Block-Update is wait-free. A Scan can only be blocked by an infinite sequence of concurrent Block-Update operations.

The implementation uses a single-writer snapshot object $H[0..k]$. All $k+1$ components of H are initially $\perp$. The i 'th component of H is used by q_i to record a list of every Update it performs, each represented by a *triple*. A triple contains a component of $\mathcal{M}$, a value, and a *timestamp* we associate with a Block-Update. For a Block-Update to r components of $\mathcal{M}$, q_i appends r triples to $H[i]$, all with the same timestamp. For clarity of presentation, we also use $n(n-1)$ unbounded arrays of single-writer registers, $L_{i,j}$, for all $i \neq j$, each indexed by non-negative integers. Each register is initially $\perp$. Only process q_i can write to $L_{i,j}[b]$ and only process q_j reads from it. The registers $L_{i,j}[b]$ are used by q_i to help q_j determine the return value of its b 'th Block-Update. As discussed in Section 2, these single-writer registers are actually implemented as part of the single-writer snapshot object provided by the real system.

```

1 procedure New-timestamp( $h$ ) by  $q_i$ 
2   for  $j \in \{0, \dots, k\}$  do
3      $t_j \leftarrow \#h_j$ 
4    $t_i \leftarrow t_i + 1$ 
5   return  $(t_0, \dots, t_k)$ 

6 procedure Get-view( $h$ ) by  $q_i$ 
7   for  $j \in \{1, \dots, m\}$  do
8     if there is an update triple in  $h$  with first
       component  $j$  then
9        $t \leftarrow \max\{t' : (j, v', t') \text{ is a triple in } h\}$ 
10      let  $(j, v, t)$  be the unique triple in  $h$  with
        component  $j$  and timestamp  $t$ 
11       $v_j \leftarrow v$ 
12    else  $v_j \leftarrow \perp$ 
13  return  $(v_1, \dots, v_m)$ 

14 procedure Scan() by  $q_i$ 
15  repeat
16     $h \leftarrow H.\text{scan}()$ 
17    for  $j \in \{0, \dots, k\} - \{i\}$  do
18       $L_{i,j}[\#h_j].\text{write}(h)$ 
19     $f \leftarrow H.\text{scan}()$ 
20  until  $h = f$ 
21  return Get-view( $h$ )

22 procedure Block-Update( $([j_1, \dots, j_c], [v_1, \dots, v_c])$ ) by  $q_i$ 
23   $h \leftarrow H.\text{scan}()$ 
24   $t \leftarrow \text{New-timestamp}(h)$ 
25   $H.\text{update}(i, h_i \cdot [(j_1, v_1, t), \dots, (j_c, v_c, t)])$ 
26   $f \leftarrow H.\text{scan}()$ 
27  for  $j \in \{0, \dots, i-1\}$  do
28     $L_{i,j}[\#f_j].\text{write}(f)$ 
    // Check for a conflicting Block-Update by a
    process with lower identifier
29   $g \leftarrow H.\text{scan}()$ 
30  if there exists  $0 \leq j < i$  such that  $\#g_j > \#h_j$  then
31    return  $\nabla$ 
32   $\text{last} \leftarrow h$ 
33  for  $j \in \{0, \dots, k\} - \{i\}$  do
34     $r[j] \leftarrow L_{j,i}[\#h_i].\text{read}()$ 
35    if  $r[j] \neq \perp$  and  $\text{last}$  is a proper prefix of  $r[j]$  then
36       $\text{last} \leftarrow r[j]$ 
37  return Get-view( $\text{last}$ )

```

Figure 2: Pseudocode for Augmented Snapshot Object Implementation.

Notation. We use upper case letters to denote instances of scan and update on H , instances of read and write on single-writer registers, and instances of Scan, Block-Update and Update on $\mathcal{M}$. The corresponding lower case letter denotes the result of a scan, read, or Block-Update operation. For example, h denotes the result of a scan H . We use h_i to denote the value of the i 'th component of h and $\#h_i$ to denote the number of Block-Update operations q_i has performed on $\mathcal{M}$, which is exactly the number of different timestamps associated with the triples recorded in h_i .

Auxiliary Procedures. A *timestamp* is a label from a partially ordered set, which can be associated with an operation, such that, if one operation completes before another operation begins, the first operation has a smaller timestamp [36]. We use a variant of vector timestamps [11, 24, 38]: Each *timestamp* is a $(k+1)$ -component vector of non-negative integers, with one component per process. Timestamps are ordered lexicographically. We use $t' > t$ to denote that timestamp t' is lexicographically larger than timestamp t and $t' \geq t$ to denote that t' is lexicographically at least as large as t .

Let h be the result of a scan. Process q_i generates a new timestamp $t = (t_0, t_1, \dots, t_k)$ from h using the locally computed function New-timestamp(h). It sets $t_j \leftarrow \#h_j$ for all $j \neq i$ and sets $t_i \leftarrow \#h_i + 1$.

For each $j \in \{1, \dots, m\}$, let v_j be the value with the lexicographically largest associated timestamp among all update triples (j, v, t) in all components of h , or $\perp$ if no such triple exists. The *view* of h , denoted view(h), is the vector $(v_1, \dots, v_m)$. It is obtained using the locally computed function Get-view(h).

Main Procedures. To perform a Scan() of $\mathcal{M}$, process q_i repeatedly performs scans of H until two consecutive results are the same. Then q_i returns the view of its last scan. Notice that Scan is not necessarily wait-free. However, it can only be blocked by an infinite sequence of Block-Update operations that modify H between every two scan operations performed by the Scan. To help other processes determine what to return from a Block-Update, q_i records the result, h , of each scan in register $L_{i,j}[\#h_j]$, for all $j \neq i$.

To perform a Block-Update($([j_1, \dots, j_c], [v_1, \dots, v_c])$) of $\mathcal{M}$, q_i first performs a scan H of H . Then it generates a timestamp, t , from the result, h , of H and appends the triples $(j_1, v_1, t), \dots, (j_c, v_c, t)$ to $H[i]$ via an update. This associates the same timestamp, t , with the Block-Update and each of the Update operations comprising it. Next, process q_i helps processes with lower identifiers by performing another scan F of H and recording its result, f , in $L_{i,j}[\#f_j]$ for all $j < i$. Then q_i performs a third scan to check whether any process with a lower identifier has performed an update after H . If so, q_i returns ∇ . This is the only way in which a Block-Update can return ∇ . Consequently, all Block-Update operations performed by q_0 are atomic.

If q_i does not return ∇ , it reads $L_{j,i}[b]$ for all $j \neq i$, where b is the number of Block-Update operations that q_i had previously performed. It determines which among these and h is the result of the latest scan and returns its view. The mechanism for determining the latest scan is described next.

If h and h' are the results of two scans of H and, if h_j is a prefix of h'_j for all $j \in \{0, 1, \dots, k\}$, we say that h is a *prefix* of h' . In addition,

if $h_j \neq h'_j$ for some $j \in \{0, 1, \dots, k\}$, we say that h is a *proper prefix* of h' . Since each update to the single-writer snapshot H appends one or more update triples to a component, the following is true.

OBSERVATION 2. *Let H and H' be scans of H with results h and h' , respectively. If H occurred before H' , then h is a prefix of h' . Conversely, if h is a proper prefix of h' , then H occurred before H' .*

Thus, by Observation 2, for any set of scans, the result of the earliest of these scans is a prefix of the result of every other scan in the set.

Linearization Points. A complete Scan operation is linearized at its last scan of H , performed on Line 19. Now consider a Block-Update, with associated timestamp t , that updates components $j_1, \dots, j_c$. For $1 \leq i \leq c$, the Update to component j_i is linearized at the first point in the execution at which H contains a triple beginning with j_i and ending with a timestamp $t' \geq t$. If multiple Update operations are linearized at the same point, then they are ordered by their associated timestamps (from earliest to latest) and then in increasing order of the components they update.

Each Update of a Block-Update, B , performed without step contention is linearized at B 's update to H on Line 25. However, it is not possible to do this for all Block-Update operations; otherwise, we would be implementing a linearizable, non-blocking augmented snapshot, which, as discussed earlier, is impossible. In our linearization, if an Update, U , that is part of B updates a component which is also updated by an Update, U' , that is part of a concurrent Block-Update by a process with a lower identifier, then U may be linearized before U' .

In the full version of the paper, we prove the following theorem.

THEOREM 3. *Figure 2 gives a non-blocking implementation of an m -component augmented snapshot object. A Block-Update by q_i returns ∇ only if its execution interval contains an update by a process q_j with $j < i$ to H on Line 25 (performed as a part of a Block-Update by q_j).*

As part of the proof, we show each Update, U , that is part of a Block-Update, B , is linearized between the scan of H in B on Line 23 and the update, u , to H in B on Line 25. To see why, suppose that U updates component j to value v and has associated timestamp t . Since u appends the triple (j, v, t) to a component of H , U is linearized no later than u . However, U could be linearized at an update, u' , prior to u . This happens if u' appends a triple (j, v', t') to a component of H , where $t' \geq t$. Note that the scan of H in B on Line 23 occurs before u' ; otherwise, the view, h , this scan returns includes (j, v', t') . Since $t = \text{New-Timestamp}(h)$, we can show that $t > t'$, which is a contradiction.

Let q_i be the process that performs B . Then $t_i = \#h_i + 1$. Since u' occurs before u , it follows that $t'_i \leq \#h_i$. But $t \geq t$ and timestamps are ordered lexicographically, so there exists $0 \leq \ell < i$ such that $t'_\ell > t_\ell$. Let g be the result of the scan of H in B on Line 29, which occurs after u and, hence, after u' . Then $\#g_\ell \geq t'_\ell > t_\ell = \#h_\ell$. This is detected by the test in B on Line 30, so B returns ∇ . Thus, all Update operations that comprise an atomic Block-Update are linearized at Line 25 of the Block-Update.

In addition, we show that different timestamps are associated with different Block-Update operations. This implies the Update

operations comprising an atomic Block-Update are linearized consecutively.

4 THE SIMULATION

In this section, we prove the following theorem:

THEOREM 4. *For $1 \leq x \leq k$, any x -obstruction-free protocol for solving k -set agreement among $n > k$ processes must use at least $\lfloor \frac{n-x}{k-x+1} \rfloor + 1$ registers.*

The standard approach for proving space complexity lower bounds is to use a *covering argument* [18]. Such an argument involves showing that it is possible to schedule the processes so that they perform block writes and cover successively more registers. The main difficulty with this approach is proving that processes do not terminate prematurely. In particular, when inserting *hidden* steps by a process before a block write, one must show that it does, in fact, write to a register that is not covered. Moreover, after a block write, one must show that the processes have not yet gained enough information to output a value and terminate.

In contrast, in our proof, we show that, if an x -obstruction-free protocol Π for k -set agreement among $n > k$ processes uses $m \leq \frac{n-x}{k-x+1}$ registers, then $k+1$ real processes can simulate an execution of Π in the simulated system. The real processes will also be solving k -set agreement in the real system. Each process q_i will initialize all the simulated processes that it is responsible for (the processes in P_i) with its own input, and will simulate their steps according to Π . It will simulate them in an order so that the resulting execution in the simulated system satisfies certain properties. Intuitively, it will be very much like performing a covering argument inside a simulation. This approach avoids the problem of processes terminating prematurely because, when a simulated process outputs a value and terminates, the real process simulating it will be able to output the same value and terminate. We will use x -obstruction-freedom of the simulated protocol, combined with properties satisfied by the simulated execution, to ensure that each real process can return a correct output in a wait-free manner.

This gives a protocol for $k+1$ real processes to solve k -set agreement in a wait-free manner, which contradicts Theorem 1. Hence, our lower bound may be viewed as a “black-box” reduction from Theorem 1.

4.1 Simulation algorithm

Fix any x -obstruction-free protocol Π for k -set agreement among $n > k$ processes that uses $m \leq \frac{n-x}{k+1-x}$ registers. As discussed in Section 2, we may assume that the protocol uses one single m -component multi-writer snapshot object, M , on which each process alternately performs scan and update, until it performs a scan on M whose result allows it to output a value and terminate.

We now describe how the real processes simulate Π . The real processes are partitioned into $k+1-x$ *covering* simulators, each responsible for m distinct simulated processes, and x *direct* simulators, each responsible for one simulated process. Notice that the total number of simulated processes is $x + m(k+1-x) \leq n$, since $m \leq \frac{n-x}{k+1-x}$.

Both types of simulators use a non-blocking implementation of an m -component augmented snapshot object, $\mathcal{M}$, for simulating the steps of processes they are responsible for. A direct simulator

directly simulates its single process in a step-by-step manner. A covering simulator attempts to simulate its set of processes so that they all cover different components of M . We note that the algorithm of a covering simulator, which we will shortly describe in detail, is similar to a covering argument, i.e. a covering simulator tries to simulate its processes so that they perform block updates and cover successively more components. This involves *locally* simulating certain steps by simulated processes, i.e. without performing any operations on $\mathcal{M}$.

We will guarantee that for each *real execution* of the simulators (i.e. an execution by the real processes in the real system) there exists a corresponding *simulated execution* of the protocol Π . However, because of the locally simulated steps, the exact correspondence between these executions is too complex to be described here without proper formalism.

Direct simulator's algorithm. A direct simulator q directly simulates its process p as follows. Initially, q sets the input of p to its input. To simulate a scan on M by p , q performs a Scan on $\mathcal{M}$. To simulate an update(j, v) on M by p , q performs a one component Block-Update($[j], [v]$) on $\mathcal{M}$, ignoring the value returned. At any point, if p outputs a value and terminates, then q outputs the same value and terminates.

A covering simulator q uses a Block-Update operation, $\mathcal{B}$, to attempt to simulate a block update by a subset of its processes. If $\mathcal{B}$ returns a view $\mathcal{V} \neq \nabla$, then q knows that $\mathcal{B}$ was *atomic*, i.e. the individual Updates in $\mathcal{B}$ can be linearized consecutively. Moreover, q knows that $\mathcal{V}$ is a view of $\mathcal{M}$ at some earlier point in the real execution such that, between this point and the first Update in $\mathcal{B}$, there are no Scan operations and no Update operations that are part of atomic Block-Update operations. At this point, q may choose to simulate *hidden steps* by one of its processes, p , effectively *revising* the past in the corresponding simulated execution. The resulting simulated execution will be indistinguishable to all other simulated processes, so p *could have* taken those steps. We emphasize that the real execution does not change. If $\mathcal{B}$ returns ∇ , then q knows that the the Update operations comprising $\mathcal{B}$ have occurred, but not necessarily consecutively. In this case, q does not use $\mathcal{B}$ to hide steps. Note that any two sequences of hidden steps are separated by at least one atomic Block-Update and, hence, do not interfere with each other.

Covering simulator's algorithm. Let $P = \{p'_1, \dots, p'_m\}$ be the set of simulated processes that a particular covering simulator q is responsible for. Initially, q sets the input of each process in P to its input. The goal of q is to simulate P such that, eventually, P covers all components of M . This corresponds to q *constructing* a Block-Update to all components of $\mathcal{M}$. In general, *constructing* a Block-Update to r components of $\mathcal{M}$ involves q simulating $p'_1, p'_2, \dots, p'_r$ until they are covering r different components of M . We describe a recursive procedure for doing so. Note that, at any point in q 's construction procedure, if a process in P outputs a value and terminates, then q outputs the same value and terminates.

As a base case, to construct a Block-Update to a single component of $\mathcal{M}$, q directly simulates p'_1 until it is about to perform an update(j_1, v_1) to M (or it terminates). If p'_1 does not terminate, then q has constructed Block-Update($[j_1], [v_1]$).

To construct a Block-Update to $r + 1$ components of $\mathcal{M}$, q constructs and performs a sequence of Block-Update operations $\mathcal{B}_1, \mathcal{B}_2, \dots$ to r components of $\mathcal{M}$. It continues until (one of its simulated processes terminates or) it constructs a Block-Update $\mathcal{B}_t$ to r components of $\mathcal{M}$ that updates the same set of components as some atomic Block-Update earlier in the sequence, i.e. for some $t' < t$, $\mathcal{B}_{t'}$ updates the same set of components as $\mathcal{B}_t$ and $\mathcal{B}_{t'}$ returns a view $\mathcal{V} \neq \nabla$. Let $j_1, \dots, j_r$ be the components that $\mathcal{B}_t$ updates and let $v_1, \dots, v_r$ be the corresponding values. After constructing $\mathcal{B}_t$, q continues its simulation of p'_{r+1} by *locally* simulating a solo execution of p'_{r+1} , assuming that the contents of M are $\mathcal{V}$ at the beginning of this execution. It does so until p'_{r+1} (terminates or is about to perform an update(j_{r+1}, v_{r+1}) to M , where $j_{r+1} \notin \{j_1, \dots, j_r\}$). This has the effect of inserting hidden steps by p'_{r+1} at the point in the simulated execution corresponding to $\mathcal{V}$ in the real execution. If none of q 's simulated processes terminate, then q has constructed Block-Update($[j_1, \dots, j_r, j_{r+1}], [v_1, \dots, v_r, v_{r+1}]$).

If q constructs a Block-Update to all m components of $\mathcal{M}$, then q locally simulates this block update by P followed by the terminating solo execution of p'_1 . This execution is applicable at any point in the future, since the block update completely overwrites M . Then process q terminates and returns the value that p'_1 returns in its simulated solo execution. This termination condition is crucial for showing that the k -set agreement algorithm for the real processes is wait-free.

4.2 Sketch of correctness proof

The formal correctness proof of the simulation is fairly technical, so we only give a sketch here. A complete proof appears in the full version of the paper.

At a high level, we would like to prove that every real execution σ corresponds to a valid execution σ of Π by the simulated processes. We would like the states of the simulated processes at the end of σ to be stored by their respective simulators at the end of σ . However, this property is not strong enough as an inductive hypothesis. In particular, when we inductively add one more step to σ , the corresponding execution σ may have to change significantly to accommodate hidden, locally simulated steps. This is a significant technical hurdle.

The implementation of $\mathcal{M}$, described in Section 3, is non-blocking and, while Block-Update operations are not necessarily atomic, the Update operations that comprise them are. This lets us guarantee that the contents of $\mathcal{M}$ throughout a real execution match the contents of M at certain configurations in the corresponding simulated execution. However, because of the revisionist nature of the simulation (i.e. the locally simulated steps by the covering simulators), there are configurations in the simulated execution in which the contents of M do not match the contents of $\mathcal{M}$ in any configuration of the real execution. This is a reason why there is no short and easy way to capture the correspondence between the real and simulated executions. It requires a complex invariant with proper formalism and a rigorous proof, which is done in the full version of the paper. In particular, we capture properties of σ that hold at points where the locally simulated steps may be inserted.

Once we have shown that σ corresponds to a valid execution σ of the protocol, we prove that the simulation is wait-free and satisfies k -agreement and validity. To see that the simulation satisfies k -agreement, notice that any value that is output by a simulator is output by a process in σ . Since the protocol satisfies k -agreement, the set of values output by the simulated processes has size at most k . Moreover, because each simulator initially sets the inputs of its simulated processes to its own input and the protocol satisfies validity, the output of a simulator must be the input of some simulator. Hence the simulation satisfies validity.

Wait-freedom. Consider any Q -only execution and any Block-Update that begins during this execution. If this Block-Update is performed by the simulator $q \in Q$ with the lowest identifier, then, by Theorem 3, it will not return ∇ . Since covering simulators have lower identifiers than direct simulators, if covering simulators continue to perform Block-Update operations, then some covering simulator, q , will continue to perform atomic Block-Update operations. The number of different sets of components that may be covered by r processes is $\binom{m}{r}$. So, to construct a Block-Update to $r + 1$ components, q needs to construct and perform at most $\binom{m}{r}$ atomic Block-Update operations to r components. Thus, q performs finitely many Block-Update operations. After q stops performing Block-Update operations, if covering simulators continue to perform Block-Update operations, then some other covering simulator will continue to perform atomic Block-Update operations. Thus, inductively, every covering simulator performs only finitely many Block-Update operations.

After covering simulators stop performing Block-Update operations, they will no longer apply update operations and, hence, cannot be observed by the x processes simulated by the x direct simulators. Since the simulated protocol is x -obstruction-free, these x processes and, hence, their direct simulators will return within a finite number of subsequent steps. Using these ideas and our invariant, we prove that no real process q_i , whether it is a covering simulator or a direct simulator, can apply infinitely many operations on $\mathcal{M}$.

If there is an infinite execution of the simulation, then some processes apply infinitely many accesses to the underlying single-writer snapshot object used to implement $\mathcal{M}$. Since our implementation of $\mathcal{M}$ is non-blocking, this means that infinitely many operations on $\mathcal{M}$ will complete. But this is impossible, since every real process applies only finitely many operations on $\mathcal{M}$. Thus the simulation is wait-free.

5 FROM NONDETERMINISTIC SOLO TERMINATION TO OBSTRUCTION-FREEDOM

A protocol is *nondeterministic solo terminating* if, for any process p and any configuration C , there exists a solo execution by p from C in which p outputs a value (and terminates) [23]. In this section, we prove the following theorem.

THEOREM 5. *If there is a nondeterministic solo terminating protocol for a task that only uses a multi-writer snapshot with m components, then there is an obstruction-free protocol for the task that only uses a multi-writer snapshot with m components.*

First, we need to formalize the notion of a nondeterministic solo terminating protocol. A nondeterministic protocol specifies a nondeterministic state machine M_p for each process p . Each state machine M_p is a 5-tuple $(S_p, v_p, \delta_p, I_p, F_p)$, where

- S_p is a set of states,
- $I_p \subseteq S_p$ is a set of initial states, one for each possible input to p ,
- $F_p \subseteq S_p$ is a set of final states, one for each possible output of p ,
- v_p specifies the next step that p will perform in each non-final state $s \in S_p - F_p$, and
- δ_p is a transition function mapping each non-final state $s \in S_p - F_p$ and possible response from step $v_p(s)$ to a nonempty subset of S_p .

A state machine is *deterministic* if δ_p maps each non-final state and possible response to a state, rather than a subset of states.

In every configuration, each process p is in some state $s \in S_p$. Initially, each process is in one of its initial states and each component of the snapshot contains $\perp$. When allocated a step by the scheduler, p does nothing if it is in a final state. If p is in state $s \in S_p - F_p$, it performs step $v_p(s)$ and, if a is the response it receives from this step, p nondeterministically chooses its next state s' from $\delta_p(s, a)$. If s' is in F_p , we consider p to have output the value corresponding to s' .

Without loss of generality, we assume that each state $s \in S_p$ contains a possible view, $E_p(s)$, of the snapshot. It is $(\perp, \dots, \perp)$ for each initial state in I_p and is updated as follows. Suppose that, when p performs $v_p(s)$, it gets response a and enters state $s' \in \delta_p(s, a)$. If $v_p(s) = \text{update}(j, v)$, then $E_p(s')$ is the same as $E_p(s)$ except component j is set to v . Otherwise, $v_p(s)$ is a scan and $E_p(s') = a$. Intuitively, $E_p(s)$ is what p expects to see in its next scan, if no other process has taken steps since its last step.

A p -solo path from a state s of length t is an alternating sequence of states in S_p and responses, $s_0, a_0, s_1, \dots, a_{t-1}, s_t$, such that $s_0 = s$, $s_t \in F_p$, and, for each $0 \leq i < t$,

- $s_{i+1} \in \delta_p(s_i, a_i)$,
- if $v_p(s_i)$ is an update, then $a_i = \text{ACK}$, and
- if $v_p(s_i)$ is a scan, then $a_i = E_p(s_i)$.

A p -solo path represents a solo terminating execution by p from a configuration C where p is in state s and $E_p(s)$ is the view of the snapshot at C .

If p is in state s in a reachable configuration C , then there exists a p -solo path from s . To see why, consider the execution σ from the initial configuration C_0 that is used to reach C . Let σ'' be the longest suffix of σ containing no scans by p and write $\sigma = \sigma' \sigma''$. Then σ' is either empty or ends with the last scan by p in σ . Let β be the subsequence of p 's steps in σ'' and let $C' = C_0 \sigma' \beta$. Note that C' is a reachable configuration of Π in which the state of p is s and the view of the snapshot is $E_p(s)$. Since Π is nondeterministic solo terminating, p has a solo terminating execution from C' and, hence, there is a p -solo path from s .

Given a nondeterministic solo terminating protocol, Π , we build a deterministic state machine $M'_p = (S_p, v_p, \delta'_p, I_p, F_p)$ from $M_p = (S_p, v_p, \delta_p, I_p, F_p)$, for each process p . The result is a deterministic protocol, Π' , which uses the same snapshot. For each state $s \in S_p - F_p$ and response a , we define $\delta'_p(s, a)$ as follows. If there is a

p -solo path from s such that the response of step $v_p(s)$ is a , then we define $\delta'_p(s, a) = s_1$, where $\sigma(s, a) = s_0, a_0, s_1, \dots, a_{t-1}, s_t$ is a shortest p -solo path with $s_0 = s$ and $a_0 = a$. Otherwise, we define $\delta'_p(s, a)$ to be any fixed state $s' \in \delta_p(s, a)$.

Correctness of Π' : Observe that every execution of Π' is an execution of Π . This is because, for each process p , each state $s \in S_p - F_p$, and each response a to $v_p(s)$, $\delta'_p(s, a) \in \delta_p(s, a)$, so $\delta'_p(s, a)$ is a state that p could be in after performing $v_p(s)$ in Π . It follows that if Π is a protocol for a task, Π' is also a protocol for the task.

Obstruction-freedom of Π' : Assume, for a contradiction, that there is an infinite solo execution α by some process p from a reachable configuration C_0 . In this execution, p never enters a final state and, so, does not output a value. Let $C_0, e_0, C_1, e_1, \dots$ be the alternating sequence of configurations and steps corresponding to α and, for $i \geq 0$, let $f_i \in S_p - F_p$ be the state of p in C_i and let a_i be the response of e_i . Since C_i is a reachable configuration in which p has state f_i , there is a p -solo path from f_i .

First, suppose that α contains only update steps, i.e. $e_i = v_p(f_i)$ is an update step, for all $i \geq 0$. Hence, there is a p -solo path from f_i where the response of $v_p(f_i)$ is $a_i = \text{ACK}$. By definition, $\sigma(f_i, a_i) = f_i, a_i, \sigma'(f_i, a_i)$, where $\sigma'(f_i, a_i)$ is a p -solo path from f_{i+1} . Since $v(f_{i+1})$ is an update step, the first response in $\sigma'(f_i, a_i)$ is $a_{i+1} = \text{ACK}$. Thus, for $i \geq 0$, the length of $\sigma(f_{i+1}, a_{i+1})$ is at most the length of $\sigma'(f_i, a_i)$, which is less than the length of $\sigma(f_i, a_i)$.

Now suppose that α contains a scan step. Let e_j be the first scan step in α . Then a_j is view of the snapshot at C_j and C_{j+1} and, by definition, $E_p(f_{j+1}) = a_j$. Since α is a solo execution by process p , $E_p(f_i)$ is the view in configuration C_i for all $i \geq j + 1$. It follows that, for each $i \geq j + 1$, f_i has a solo path where the response of step $v_p(f_i)$ is a_i . By definition, $\sigma(f_i, a_i) = f_i, a_i, \sigma'(f_i, a_i)$, where $\sigma'(f_i, a_i)$ is a p -solo path from f_{i+1} . Since $E_p(f_i)$ is the view of the snapshot at C_i , $\sigma(f_i, a_i)$ represents a solo execution from C_i , so each response following a state s in $\sigma(f_i, a_i)$ matches the response from the corresponding step $v_p(s)$ in the solo execution. In particular, the first response in $\sigma'(f_i, a_i)$ is a_{i+1} . Thus, for $i \geq j + 1$, the length of $\sigma(f_{i+1}, a_{i+1})$ is at most the length of $\sigma'(f_i, a_i)$, which is less than the length of $\sigma(f_i, a_i)$.

In both cases, there is a $k \geq 0$ such that, for all $i \geq k$, the length of $\sigma(f_i, a_i)$ is decreasing. Let t be the length of $\sigma(f_k, a_k)$. Then there exists $k' \leq k + t$ such that the length of $\sigma(f_{k'}, a_{k'})$ is 0. Hence, $f_{k'} \in F_p$. This is a contradiction.

6 CONCLUSIONS AND FUTURE WORK

We conjecture that the space complexity of x -obstruction-free k -set agreement is $n - k + x$, matching the upper bound of [16]. Our paper makes significant progress by proving the first non-constant lower bound for non-anonymous processes. This lower bound is asymptotically tight when k and x are constant and tight when $x = 1$ and either $k = 1$ (obstruction-free consensus) and $k = n - 1$ (obstruction-free $(n - 1)$ -set agreement). Equally importantly, our simulation technique uses a new approach. It is conceivable that this approach can be extended to obtain a tight lower bound when $k > 1$ and $1 \leq x \leq k$.

Extending the techniques introduced in this paper and applying them to other problems may be fruitful. We demonstrate this in the full version of the paper by applying the simulation with

minor modifications to get a space lower bound for approximate agreement that is within a factor of 2 of optimal. It is possible to generalize our approach to obtain space lower bounds for a class of *simulatable* tasks, which are either not deterministically solvable in a wait-free manner or have large lower bounds on their step complexity. It would be great to generalize our approach to a larger class of tasks.

We proved that a space lower bound for obstruction-free protocols implies a space lower bound for protocols that satisfy non-deterministic solo termination (including randomized wait-free protocols). by converting any nondeterministic solo terminating protocol to an obstruction-free protocol that uses the same number of registers. This allows researchers to focus on deriving space lower bounds for obstruction-free protocols.

It is known how to convert any deterministic obstruction-free protocol for n processes that has solo step complexity b into a randomized wait-free protocol against an oblivious adversary, which has expected step complexity polynomial in n and b [29]. However, our construction provides no bound on the solo step complexity of the resulting obstruction-free protocol. It would be interesting to improve the construction to bound the solo step complexity of the resulting protocols.

ACKNOWLEDGEMENTS

Part of this work was performed when Rati Gelashvili was a Ph.D. student at MIT. This work was supported by an NSERC Discovery Grant, an NSERC Postgraduate Scholarship, a University of Toronto Faculty of Arts & Science Postdoctoral Fellowship, NSF grants IIS-1607189, IIS-1447786, and CCF1563880, a donation from SUN Microsystems, and a grant from Intel Corporation.

REFERENCES

- [1] Karl Abrahamson. 1988. On achieving consensus using a shared memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*. 291–302.
- [2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *J. ACM* 40, 4 (1993), 873–890.
- [3] James Aspnes and Maurice Herlihy. 1990. Fast randomized consensus using shared memory. *Journal of Algorithms* 11, 3 (1990), 441–461.
- [4] Hagit Attiya and Armando Castañeda. 2013. A non-topological proof for the impossibility of k -set agreement. *Theoretical Computer Science* 512 (2013), 41–48.
- [5] Hagit Attiya and Keren Censor. 2008. Tight bounds for asynchronous randomized consensus. *J. ACM* 55, 5 (2008), 20.
- [6] Hagit Attiya and Faith Ellen. 2014. Impossibility Results for Distributed Computing. *Synthesis Lectures on Distributed Computing Theory* 5, 1 (2014), 1–162.
- [7] Hagit Attiya, Alla Gorbach, and Shlomo Moran. 2002. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation* 173, 2 (2002), 162–183.
- [8] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. 2005. Computing with Reads and Writes in the Absence of Step Contention. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC '05)*. 122–136.
- [9] Hagit Attiya, Nancy Lynch, and Nir Shavit. 1994. Are wait-free algorithms fast? *J. ACM* 41, 4 (1994), 725–763.
- [10] Hagit Attiya and Ami Paz. 2016. Counting-based Impossibility Proofs for Set Agreement and Renaming. *J. Parallel and Distrib. Comput.* 87, C (2016), 1–12.
- [11] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, USA.
- [12] Paul Beame. 1986. Limits on the power of concurrent-write parallel machines. In *Proceedings of the 18th ACM Symposium on Theory of Computing (STOC '86)*. 169–176.
- [13] Michael Ben-Or. 1983. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing (PODC '83)*. 27–30.

- [14] Elizabeth Borowsky and Eli Gafni. 1993. Generalized FLP impossibility result for t -resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC '93)*. 91–100.
- [15] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. 2001. The BG distributed simulation algorithm. *Distributed Computing* 14, 3 (2001), 127–146.
- [16] Zohir Bouzid, Michel Raynal, and Pierre Sutra. 2015. Anonymous obstruction-free (n, k) -set agreement with $n - k + 1$ atomic read/write registers. *Distributed Computing* 31, 2 (2015), 1–19.
- [17] Jack R. Bowman. 2011. *Obstruction-free Snapshot, Obstruction-free Consensus, and Fetch-and-add Modulo k*. Technical Report TR2011-681. Dartmouth College.
- [18] James E Burns and Nancy A Lynch. 1993. Bounds on shared memory for mutual exclusion. *Information and Computation* 107, 2 (1993), 171–184.
- [19] Soma Chaudhuri. 1993. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation* 105, 1 (1993), 132–158.
- [20] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. 2013. Black Art: Obstruction-free k -set agreement with $|MWMR \text{ registers}| < |processes|$. In *Proceedings of the 1st International Conference on Networked Systems (NETYS '13)*. 28–41.
- [21] Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. 2015. On the Space Complexity of Set Agreement. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC '15)*. 271–280.
- [22] Danny Dolev, Nancy Lynch, Shlomit Pinter, Eugene Stark, and William Weihl. 1986. Reaching approximate agreement in the presence of faults. *J. ACM* 33, 3 (1986), 499–516.
- [23] Faith Fich, Maurice Herlihy, and Nir Shavit. 1998. On the space complexity of randomized synchronization. *J. ACM* 45, 5 (1998), 843–862.
- [24] Colin Fidge. 1991. Logical time in distributed computing systems. *Computer* 24, 8 (1991), 28–33.
- [25] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (1985), 374–382.
- [26] Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. 2016. Read-Write Memory and k -Set Consensus as an Affine Task. In *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS '16)*. 6:1–6:17.
- [27] Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. 2014. A Generalized Asynchronous Computability Theorem. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC '14)*. 222–231.
- [28] Rati Gelashvili. 2015. On the optimal space complexity of consensus for anonymous processes. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC '15)*. 452–466.
- [29] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. 2013. An $O(\sqrt{n})$ Space Bound for Obstruction-Free Leader Election. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC '13)*. 46–60.
- [30] Rachid Guerraoui and Eric Ruppert. 2005. What can be implemented anonymously?. In *Proceedings of the 19th International Conference on Distributed Computing (DISC '05)*. 244–259.
- [31] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (1991), 124–149.
- [32] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. 2013. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc.
- [33] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*. 522–.
- [34] Maurice Herlihy and Nir Shavit. 1999. The topological structure of asynchronous computability. *J. ACM* 46, 6 (1999), 858–923.
- [35] Maurice Herlihy and Jeannette Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.
- [36] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
- [37] M. Loui and H. Abu-Amara. 1987. Memory Requirements for Agreement among Unreliable Asynchronous Processes. In *Advances in Computing Research*. Vol. 4. JAI Press, Inc., 163–183.
- [38] Friedemann Mattern. 1989. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms* 1, 23 (1989), 215–226.
- [39] Michael Saks, Nir Shavit, and Heather Woll. 1991. Optimal Time Randomized Consensus - Making Resilient Algorithms Fast in Practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '91)*. 351–362.
- [40] Michael Saks and Fotios Zaharoglou. 2000. Wait-free k -set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.* 29, 5 (2000), 1449–1483.
- [41] Vikram Saraph, Maurice Herlihy, and Eli Gafni. 2016. Asynchronous Computability Theorems for t -Resilient Systems. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC '16)*. 428–441.
- [42] Lefschetz Solomon. 1949. *Introduction to Topology*. Princeton University Press, New Jersey, United States.
- [43] Gadi Taubenfeld. 2017. Contention-sensitive data structures and algorithms. *Theoretical Computer Science* 677 (2017), 41–55.
- [44] Jiong Yang, Gil Neiger, and Eli Gafni. 1998. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC '98)*. 297–306.
- [45] Leqi Zhu. 2015. Brief Announcement: Tight space bounds for memoryless anonymous consensus. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC '15)*. 665.
- [46] Leqi Zhu. 2016. A tight space bound for consensus. In *Proceedings of the 48th ACM Symposium on Theory of Computing (STOC '16)*. 345–350.